BOAZ BARAK

INTRODUCTION TO THEORETICAL COMPUTER SCIENCE

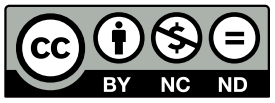
TEXTBOOK IN PREPARATION.
AVAILABLE ON HTTPS://INTROTCS.ORG

Text available on https://github.com/boazbk/tcs - please post any issues there - thank you!

This version was compiled on Monday 19th December, 2022 22:58

Copyright © 2022 Boaz Barak

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.



To Ravit, Alma and Goren.

Contents

Preface 9

Preliminaries 17

0 Introduction 19

1 Mathematical Background 37

2 Computation and Representation 73

I Finite computation 111

3 Defining computation 113

4 Syntactic sugar, and computing every function 149

5 Code as data, data as code 175

II Uniform computation 205

6 Functions with Infinite domains, Automata, and Regular expressions 207

7 Loops and infinity 241

8 Equivalent models of computation 271

9 Universality and uncomputability 315

10 Restricted computational models 347

11 Is every theorem provable? 365

III Efficient algorithms 385

12 Efficient computation: An informal introduction 387

13 Modeling running time 407

14 Polynomial-time reductions 441

15 NP, NP completeness, and the Cook-Levin Theorem 469

16 What if P equals NP? 489

17 Space bounded computation 509

IV Randomized computation 511

18 Probability Theory 101 513

19 Probabilistic computation 533

20 Modeling randomized computation 545

V Advanced topics 569

21 Cryptography 571

22 Proofs and algorithms 599

23 Quantum computing 601

VI Appendices 631

Contents (detailed)

Preface **9**
- 0.1 To the student . . . 10
  - 0.1.1 Is the effort worth it? . . . 11
- 0.2 To potential instructors . . . 12
- 0.3 Acknowledgements . . . 14

Preliminaries **17**

0 Introduction **19**
- 0.1 Integer multiplication: an example of an algorithm . . . 20
- 0.2 Extended Example: A faster way to multiply (optional) 22
- 0.3 Algorithms beyond arithmetic . . . 27
- 0.4 On the importance of negative results . . . 28
- 0.5 Roadmap to the rest of this book . . . 29
  - 0.5.1 Dependencies between chapters . . . 30
- 0.6 Exercises . . . 32
- 0.7 Bibliographical notes . . . 33

1 Mathematical Background **37**
- 1.1 This chapter: a reader's manual . . . 37
- 1.2 A quick overview of mathematical prerequisites . . . 38
- 1.3 Reading mathematical texts . . . 39
  - 1.3.1 Definitions . . . 40
  - 1.3.2 Assertions: Theorems, lemmas, claims . . . 40
  - 1.3.3 Proofs . . . 40
- 1.4 Basic discrete math objects . . . 41
  - 1.4.1 Sets . . . 41
  - 1.4.2 Special sets . . . 42
  - 1.4.3 Functions . . . 44
  - 1.4.4 Graphs . . . 46
  - 1.4.5 Logic operators and quantifiers . . . 49
  - 1.4.6 Quantifiers for summations and products . . . 50
  - 1.4.7 Parsing formulas: bound and free variables . . . 50
  - 1.4.8 Asymptotics and Big-O notation . . . 52

1.4.9 Some "rules of thumb" for Big-O notation 53
1.5 Proofs 54
1.5.1 Proofs and programs 55
1.5.2 Proof writing style 55
1.5.3 Patterns in proofs 56
1.6 Extended example: Topological Sorting 59
1.6.1 Mathematical induction 60
1.6.2 Proving the result by induction 61
1.6.3 Minimality and uniqueness 63
1.7 This book: notation and conventions 65
1.7.1 Variable name conventions 66
1.7.2 Some idioms 67
1.8 Exercises 69
1.9 Bibliographical notes 71

2 Computation and Representation **73**
2.1 Defining representations 75
2.1.1 Representing natural numbers 76
2.1.2 Meaning of representations (discussion) 78
2.2 Representations beyond natural numbers 78
2.2.1 Representing (potentially negative) integers . . . 79
2.2.2 Two's complement representation (optional) . . 79
2.2.3 Rational numbers and representing pairs of strings 80
2.3 Representing real numbers 82
2.4 Cantor's Theorem, countable sets, and string representations of the real numbers 83
2.4.1 Corollary: Boolean functions are uncountable . . 89
2.4.2 Equivalent conditions for countability 89
2.5 Representing objects beyond numbers 90
2.5.1 Finite representations 91
2.5.2 Prefix-free encoding 91
2.5.3 Making representations prefix-free 94
2.5.4 "Proof by Python" (optional) 95
2.5.5 Representing letters and text 97
2.5.6 Representing vectors, matrices, images 99
2.5.7 Representing graphs 99
2.5.8 Representing lists and nested lists 99
2.5.9 Notation 100
2.6 Defining computational tasks as mathematical functions 100
2.6.1 Distinguish functions from programs! 102
2.7 Exercises 104
2.8 Bibliographical notes 108

I Finite computation **111**

3 Defining computation **113**
3.1 Defining computation . . . 115
3.2 Computing using AND, OR, and NOT. . . . 116
3.2.1 Some properties of AND and OR . . . 118
3.2.2 Extended example: Computing *XOR* from *AND*, *OR*, and *NOT* . . . 119
3.2.3 Informally defining "basic operations" and "algorithms" . . . 121
3.3 Boolean Circuits . . . 123
3.3.1 Boolean circuits: a formal definition . . . 124
3.4 Straight-line programs . . . 127
3.4.1 Specification of the AON-CIRC programming language . . . 128
3.4.2 Proving equivalence of AON-CIRC programs and Boolean circuits . . . 130
3.5 Physical implementations of computing devices (digression) . . . 132
3.5.1 Transistors . . . 132
3.5.2 Logical gates from transistors . . . 133
3.5.3 Biological computing . . . 133
3.5.4 Cellular automata and the game of life . . . 134
3.5.5 Neural networks . . . 134
3.5.6 A computer made from marbles and pipes . . . 135
3.6 The NAND function . . . 135
3.6.1 NAND Circuits . . . 136
3.6.2 More examples of NAND circuits (optional) . . . 138
3.6.3 The NAND-CIRC Programming language . . . 139
3.7 Equivalence of all these models . . . 141
3.7.1 Circuits with other gate sets . . . 142
3.7.2 Specification vs. implementation (again) . . . 143
3.8 Exercises . . . 145
3.9 Biographical notes . . . 147

4 Syntactic sugar, and computing every function **149**
4.1 Some examples of syntactic sugar . . . 151
4.1.1 User-defined procedures . . . 151
4.1.2 Proof by Python (optional) . . . 153
4.1.3 Conditional statements . . . 154
4.2 Extended example: Addition and Multiplication (optional) . . . 157
4.3 The LOOKUP function . . . 158

4.3.1 Constructing a NAND-CIRC program for *LOOKUP* . . . 159
4.4 Computing *every* function . . . 161
4.4.1 Proof of NAND's Universality . . . 162
4.4.2 Improving by a factor of n (optional) . . . 163
4.5 Computing every function: An alternative proof . . . 165
4.6 The class $SIZE_{n,m}(s)$ . . . 167
4.7 Exercises . . . 170
4.8 Bibliographical notes . . . 174

5 Code as data, data as code **175**
5.1 Representing programs as strings . . . 177
5.2 Counting programs, and lower bounds on the size of NAND-CIRC programs . . . 178
5.2.1 Size hierarchy theorem (optional) . . . 180
5.3 The tuples representation . . . 182
5.3.1 From tuples to strings . . . 183
5.4 A NAND-CIRC interpreter in NAND-CIRC . . . 184
5.4.1 Efficient universal programs . . . 185
5.4.2 A NAND-CIRC interpeter in "pseudocode" . . . 186
5.4.3 A NAND interpreter in Python . . . 188
5.4.4 Constructing the NAND-CIRC interpreter in NAND-CIRC . . . 188
5.5 A Python interpreter in NAND-CIRC (discussion) . . . 191
5.6 The physical extended Church-Turing thesis (discussion) . . . 193
5.6.1 Attempts at refuting the PECTT . . . 195
5.7 Recap of Part I: Finite Computation . . . 200
5.8 Exercises . . . 201
5.9 Bibliographical notes . . . 203

II Uniform computation **205**

6 Functions with Infinite domains, Automata, and Regular expressions **207**
6.1 Functions with inputs of unbounded length . . . 208
6.1.1 Varying inputs and outputs . . . 209
6.1.2 Formal Languages . . . 211
6.1.3 Restrictions of functions . . . 211
6.2 Deterministic finite automata (optional) . . . 212
6.2.1 Anatomy of an automaton (finite vs. unbounded) 215
6.2.2 DFA-computable functions . . . 216
6.3 Regular expressions . . . 217
6.3.1 Algorithms for matching regular expressions . . . 221

6.4 Efficient matching of regular expressions (optional) . . 223
6.4.1 Matching regular expressions using DFAs 227
6.4.2 Equivalence of regular expressions and automata 228
6.4.3 Closure properties of regular expressions 230
6.5 Limitations of regular expressions and the pumping lemma 231
6.6 Answering semantic questions about regular expressions 236
6.7 Exercises 239
6.8 Bibliographical notes 240

7 Loops and infinity **241**
7.1 Turing Machines 242
7.1.1 Extended example: A Turing machine for palindromes 244
7.1.2 Turing machines: a formal definition 245
7.1.3 Computable functions 247
7.1.4 Infinite loops and partial functions 248
7.2 Turing machines as programming languages 249
7.2.1 The NAND-TM Programming language 251
7.2.2 Sneak peak: NAND-TM vs Turing machines . . . 254
7.2.3 Examples 255
7.3 Equivalence of Turing machines and NAND-TM programs 257
7.3.1 Specification vs implementation (again) 260
7.4 NAND-TM syntactic sugar 261
7.4.1 "GOTO" and inner loops 261
7.5 Uniformity, and NAND vs NAND-TM (discussion) . . 263
7.6 Exercises 265
7.7 Bibliographical notes 267

8 Equivalent models of computation **271**
8.1 RAM machines and NAND-RAM 273
8.2 The gory details (optional) 277
8.2.1 Indexed access in NAND-TM 277
8.2.2 Two dimensional arrays in NAND-TM 279
8.2.3 All the rest 279
8.3 Turing equivalence (discussion) 280
8.3.1 The "Best of both worlds" paradigm 281
8.3.2 Let's talk about abstractions 281
8.3.3 Turing completeness and equivalence, a formal definition (optional) 283
8.4 Cellular automata 284

8.4.1 One dimensional cellular automata are Turing complete . . . 286
8.4.2 Configurations of Turing machines and the next-step function . . . 287
8.5 Lambda calculus and functional programming languages . . . 290
8.5.1 Applying functions to functions . . . 290
8.5.2 Obtaining multi-argument functions via Currying 291
8.5.3 Formal description of the λ calculus . . . 292
8.5.4 Infinite loops in the λ calculus . . . 295
8.6 The "Enhanced" λ calculus . . . 295
8.6.1 Computing a function in the enhanced λ calculus 298
8.6.2 Enhanced λ calculus is Turing-complete . . . 298
8.7 From enhanced to pure λ calculus . . . 301
8.7.1 List processing . . . 302
8.7.2 The Y combinator, or recursion without recursion 303
8.8 The Church-Turing Thesis (discussion) . . . 306
8.8.1 Different models of computation . . . 307
8.9 Exercises . . . 308
8.10 Bibliographical notes . . . 312

9 Universality and uncomputability 315
9.1 Universality or a meta-circular evaluator . . . 316
9.1.1 Proving the existence of a universal Turing Machine . . . 318
9.1.2 Implications of universality (discussion) . . . 320
9.2 Is every function computable? . . . 321
9.3 The Halting problem . . . 323
9.3.1 Is the Halting problem really hard? (discussion) 326
9.3.2 A direct proof of the uncomputability of *HALT* (optional) . . . 327
9.4 Reductions . . . 329
9.4.1 Example: Halting on the zero problem . . . 330
9.5 Rice's Theorem and the impossibility of general software verification . . . 334
9.5.1 Rice's Theorem . . . 335
9.5.2 Halting and Rice's Theorem for other Turing-complete models . . . 340
9.5.3 Is software verification doomed? (discussion) . . 341
9.6 Exercises . . . 342
9.7 Bibliographical notes . . . 345

10 Restricted computational models 347
10.1 Turing completeness as a bug . . . 347

10.2 Context free grammars . . . 349
10.2.1 Context-free grammars as a computational model 351
10.2.2 The power of context free grammars . . . 353
10.2.3 Limitations of context-free grammars (optional) 355
10.3 Semantic properties of context free languages . . . 357
10.3.1 Uncomputability of context-free grammar equivalence (optional) . . . 357
10.4 Summary of semantic properties for regular expressions and context-free grammars . . . 360
10.5 Exercises . . . 361
10.6 Bibliographical notes . . . 362

11 Is every theorem provable? 365
11.1 Hilbert's Program and Gödel's Incompleteness Theorem 366
11.1.1 Defining "Proof Systems" . . . 367
11.2 Gödel's Incompleteness Theorem: Computational variant . . . 369
11.3 Quantified integer statements . . . 371
11.4 Diophantine equations and the MRDP Theorem . . . 373
11.5 Hardness of quantified integer statements . . . 374
11.5.1 Step 1: Quantified mixed statements and computation histories . . . 375
11.5.2 Step 2: Reducing mixed statements to integer statements . . . 378
11.6 Exercises . . . 380
11.7 Bibliographical notes . . . 383

III Efficient algorithms 385

12 Efficient computation: An informal introduction 387
12.1 Problems on graphs . . . 389
12.1.1 Finding the shortest path in a graph . . . 390
12.1.2 Finding the longest path in a graph . . . 392
12.1.3 Finding the minimum cut in a graph . . . 392
12.1.4 Min-Cut Max-Flow and Linear programming . . . 393
12.1.5 Finding the maximum cut in a graph . . . 395
12.1.6 A note on convexity . . . 395
12.2 Beyond graphs . . . 397
12.2.1 SAT . . . 397
12.2.2 Solving linear equations . . . 398
12.2.3 Solving quadratic equations . . . 399
12.3 More advanced examples . . . 399
12.3.1 Determinant of a matrix . . . 399
12.3.2 Permanent of a matrix . . . 401

12.3.3 Finding a zero-sum equilibrium 401
12.3.4 Finding a Nash equilibrium 402
12.3.5 Primality testing 402
12.3.6 Integer factoring 403
12.4 Our current knowledge 403
12.5 Exercises 404
12.6 Bibliographical notes 404
12.7 Further explorations 405

13 Modeling running time **407**
13.1 Formally defining running time 409
13.1.1 Polynomial and Exponential Time 410
13.2 Modeling running time using RAM Machines / NAND-RAM 412
13.3 Extended Church-Turing Thesis (discussion) 417
13.4 Efficient universal machine: a NAND-RAM interpreter in NAND-RAM 418
13.4.1 Timed Universal Turing Machine 420
13.5 The time hierarchy theorem 421
13.6 Non-uniform computation 425
13.6.1 Oblivious NAND-TM programs 427
13.6.2 "Unrolling the loop": algorithmic transformation of Turing Machines to circuits 430
13.6.3 Can uniform algorithms simulate non-uniform ones? 432
13.6.4 Uniform vs. Non-uniform computation: A recap 434
13.7 Exercises 435
13.8 Bibliographical notes 438

14 Polynomial-time reductions **441**
14.1 Formal definitions of problems 442
14.2 Polynomial-time reductions 443
14.2.1 Whistling pigs and flying horses 444
14.3 Reducing 3SAT to zero one and quadratic equations . . 446
14.3.1 Quadratic equations 449
14.4 The subset sum problem 451
14.5 The independent set problem 453
14.6 Some exercises and anatomy of a reduction. 456
14.6.1 Dominating set 457
14.6.2 Anatomy of a reduction 460
14.7 Reducing Independent Set to Maximum Cut 462
14.8 Reducing 3SAT to Longest Path 464
14.8.1 Summary of relations 466
14.9 Exercises 467

14.10 Bibliographical notes 467

15 NP, NP completeness, and the Cook-Levin Theorem 469
15.1 The class **NP** 471
15.1.1 Examples of functions in **NP** 473
15.1.2 Basic facts about **NP** 474
15.2 From **NP** to 3SAT: The Cook-Levin Theorem 476
15.2.1 What does this mean? 477
15.2.2 The Cook-Levin Theorem: Proof outline 478
15.3 The *NANDSAT* Problem, and why it is **NP** hard 479
15.4 The *3NAND* problem 481
15.5 From *3NAND* to *3SAT* 483
15.6 Wrapping up 484
15.7 Exercises 486
15.8 Bibliographical notes 487

16 What if P equals NP? 489
16.1 Search-to-decision reduction 491
16.2 Optimization 493
16.2.1 Example: Supervised learning 496
16.2.2 Example: Breaking cryptosystems 497
16.3 Finding mathematical proofs 497
16.4 Quantifier elimination (advanced) 499
16.4.1 Application: self improving algorithm for *3SAT* . 501
16.5 Approximating counting problems and posterior sampling (advanced, optional) 502
16.6 What does all of this imply? 503
16.7 Can $\mathbf{P} \neq \mathbf{NP}$ be neither true nor false? 505
16.8 Is $\mathbf{P} = \mathbf{NP}$ "in practice"? 506
16.9 What if $\mathbf{P} \neq \mathbf{NP}$? 507
16.10 Exercises 508
16.11 Bibliographical notes 508

17 Space bounded computation 509
17.1 Exercises 509
17.2 Bibliographical notes 509

IV Randomized computation 511

18 Probability Theory 101 513
18.1 Random coins 514
18.1.1 Random variables 517
18.1.2 Distributions over strings 519
18.1.3 More general sample spaces 519

18.2 Correlations and independence . . . 520
18.2.1 Independent random variables . . . 521
18.2.2 Collections of independent random variables . . . 523
18.3 Concentration and tail bounds . . . 523
18.3.1 Chebyshev's Inequality . . . 525
18.3.2 The Chernoff bound . . . 526
18.3.3 Application: Supervised learning and empirical risk minimization . . . 527
18.4 Exercises . . . 529
18.5 Bibliographical notes . . . 532

19 Probabilistic computation **533**
19.1 Finding approximately good maximum cuts . . . 534
19.1.1 Amplifying the success of randomized algorithms 535
19.1.2 Success amplification . . . 536
19.1.3 Two-sided amplification . . . 537
19.1.4 What does this mean? . . . 538
19.2 Solving SAT through randomization . . . 539
19.3 Bipartite matching . . . 540
19.4 Exercises . . . 543
19.5 Bibliographical notes . . . 544
19.6 Acknowledgements . . . 544

20 Modeling randomized computation **545**
20.1 Modeling randomized computation . . . 546
20.1.1 An alternative view: random coins as an "extra input" . . . 549
20.1.2 Success amplification of two-sided error algorithms . . . 551
20.2 **BPP** and **NP** completeness . . . 552
20.3 The power of randomization . . . 553
20.3.1 Solving **BPP** in exponential time . . . 553
20.3.2 Simulating randomized algorithms by circuits . . . 554
20.4 Derandomization . . . 555
20.4.1 Pseudorandom generators . . . 557
20.4.2 From existence to constructivity . . . 558
20.4.3 Usefulness of pseudorandom generators . . . 559
20.5 **P** = **NP** and **BPP** vs **P** . . . 560
20.6 Non-constructive existence of pseudorandom generators (advanced, optional) . . . 563
20.7 Exercises . . . 566
20.8 Bibliographical notes . . . 566

V Advanced topics **569**

21 Cryptography **571**
21.1 Classical cryptosystems . . . 572
21.2 Defining encryption . . . 574
21.3 Defining security of encryption . . . 575
21.4 Perfect secrecy . . . 577
21.4.1 Example: Perfect secrecy in the battlefield . . . 578
21.4.2 Constructing perfectly secret encryption . . . 579
21.5 Necessity of long keys . . . 581
21.6 Computational secrecy . . . 582
21.6.1 Stream ciphers or the "derandomized one-time pad" . . . 584
21.7 Computational secrecy and **NP** . . . 587
21.8 Public key cryptography . . . 589
21.8.1 Defining public key encryption . . . 591
21.8.2 Diffie-Hellman key exchange . . . 592
21.9 Other security notions . . . 594
21.10 Magic . . . 594
21.10.1 Zero knowledge proofs . . . 595
21.10.2 Fully homomorphic encryption . . . 595
21.10.3 Multiparty secure computation . . . 596
21.11 Exercises . . . 597
21.12 Bibliographical notes . . . 597

22 Proofs and algorithms **599**
22.1 Exercises . . . 599
22.2 Bibliographical notes . . . 599

23 Quantum computing **601**
23.1 The double slit experiment . . . 602
23.2 Quantum amplitudes . . . 602
23.3 Bell's Inequality . . . 605
23.4 Quantum weirdness . . . 606
23.5 Quantum computing and computation - an executive summary. . . . 607
23.6 Quantum systems . . . 609
23.6.1 Quantum amplitudes . . . 610
23.6.2 Recap . . . 611
23.7 Analysis of Bell's Inequality (optional) . . . 612
23.8 Quantum computation . . . 614
23.8.1 Quantum circuits . . . 615
23.8.2 QNAND-CIRC programs (optional) . . . 617
23.8.3 Uniform computation . . . 618
23.9 Physically realizing quantum computation . . . 619

23.10 Shor's Algorithm: Hearing the shape of prime factors . 620
23.10.1 Period finding 621
23.10.2 Shor's Algorithm: A bird's eye view 621
23.11 Quantum Fourier Transform (advanced, optional) . . . 624
23.11.1 Quantum Fourier Transform over the Boolean Cube: Simon's Algorithm 625
23.11.2 From Fourier to Period finding: Simon's Algorithm (advanced, optional) 626
23.11.3 From Simon to Shor (advanced, optional) 627
23.12 Exercises 628
23.13 Bibliographical notes 629
23.14 Further explorations 630
23.15 Acknowledgements 630

VI Appendices **631**

Preface

> *"We make ourselves no promises, but we cherish the hope that the unobstructed pursuit of useless knowledge will prove to have consequences in the future as in the past" ... "An institution which sets free successive generations of human souls is amply justified whether or not this graduate or that makes a so-called useful contribution to human knowledge. A poem, a symphony, a painting, a mathematical truth, a new scientific fact, all bear in themselves all the justification that universities, colleges, and institutes of research need or require"*, Abraham Flexner, The Usefulness of Useless Knowledge, 1939.

> *"I suggest that you take the hardest courses that you can, because you learn the most when you challenge yourself... CS 121 I found pretty hard."*, Mark Zuckerberg, 2005.

This is a textbook for an undergraduate introductory course on theoretical computer science. The educational goals of this book are to convey the following:

- That computation arises in a variety of natural and human-made systems, and not only in modern silicon-based computers.
- Similarly, beyond being an extremely important *tool*, computation also serves as a useful *lens* to describe natural, physical, mathematical and even social concepts.
- The notion of *universality* of many different computational models, and the related notion of the duality between *code* and *data*.
- The idea that one can precisely define a mathematical model of computation, and then use that to prove (or sometimes only conjecture) lower bounds and impossibility results.
- Some of the surprising results and discoveries in modern theoretical computer science, including the prevalence of NP-completeness, the power of interaction, the power of randomness on one hand and the possibility of derandomization on the other, the ability to use hardness "for good" in cryptography, and the fascinating possibility of quantum computing.

I hope that following this course, students would be able to recognize computation, with both its power and pitfalls, as it arises in various settings, including seemingly "static" content or "restricted" formalisms such as macros and scripts. They should be able to follow through the logic of *proofs* about computation, including the central concept of a *reduction*, as well as understanding "self-referential" proofs (such as diagonalization-based proofs that involve programs given their own code as input). Students should understand that some problems are *inherently intractable*, and be able to recognize the potential for intractability when they are faced with a new problem. While this book only touches on cryptography, students should understand the basic idea of how we can use computational hardness for cryptographic purposes. However, more than any specific skill, this book aims to introduce students to a new way of thinking of computation as an object in its own right and to illustrate how this new way of thinking leads to far-reaching insights and applications.

My aim in writing this text is to try to convey these concepts in the simplest possible way and try to make sure that the formal notation and model help elucidate, rather than obscure, the main ideas. I also tried to take advantage of modern students' familiarity (or at least interest!) in programming, and hence use (highly simplified) programming languages to describe our models of computation. That said, this book does not assume fluency with any particular programming language, but rather only some familiarity with the general *notion* of programming. We will use programming metaphors and idioms, occasionally mentioning specific programming languages such as *Python*, *C*, or *Lisp*, but students should be able to follow these descriptions even if they are not familiar with these languages.

Proofs in this book, including the existence of a universal Turing Machine, the fact that every finite function can be computed by some circuit, the Cook-Levin theorem, and many others, are often constructive and algorithmic, in the sense that they ultimately involve transforming one program to another. While it is possible to follow these proofs without seeing the code, I do think that having access to the code, and the ability to play around with it and see how it acts on various programs, can make these theorems more concrete for the students. To that end, an accompanying website (which is still a work in progress) allows executing programs in the various computational models we define, as well as seeing constructive proofs of some of the theorems.

0.1 TO THE STUDENT

This book can be challenging, mainly because it brings together a variety of ideas and techniques in the study of computation. There

are quite a few technical hurdles to master, whether it is following the diagonalization argument for proving the Halting Problem is undecidable, combinatorial gadgets in NP-completeness reductions, analyzing probabilistic algorithms, or arguing about the adversary to prove the security of cryptographic primitives.

The best way to engage with this material is to read these notes **actively**, so make sure you have a pen ready. While reading, I encourage you to stop and think about the following:

- When I state a theorem, stop and take a shot at proving it on your own *before* reading the proof. You will be amazed by how much better you can understand a proof even after only 5 minutes of attempting it on your own.
- When reading a definition, make sure that you understand what the definition means, and what the natural examples are of objects that satisfy it and objects that do not. Try to think of the motivation behind the definition, and whether there are other natural ways to formalize the same concept.
- Actively notice which questions arise in your mind as you read the text, and whether or not they are answered in the text.

As a general rule, it is more important that you understand the **definitions** than the **theorems**, and it is more important that you understand a **theorem statement** than its **proof**. After all, before you can prove a theorem, you need to understand what it states, and to understand what a theorem is about, you need to know the definitions of the objects involved. Whenever a proof of a theorem is at least somewhat complicated, I provide a "proof idea." Feel free to skip the actual proof in a first reading, focusing only on the proof idea.

This book contains some code snippets, but this is by no means a programming text. You don't need to know how to program to follow this material. The reason we use code is that it is a *precise* way to describe computation. Particular implementation details are not as important to us, and so we will emphasize code readability at the expense of considerations such as error handling, encapsulation, etc. that can be extremely important for real-world programming.

0.1.1 Is the effort worth it?

This is not an easy book, and you might reasonably wonder why should you spend the effort in learning this material. A traditional justification for a "Theory of Computation" course is that you might encounter these concepts later on in your career. Perhaps you will come across a hard problem and realize it is NP complete, or find a need to use what you learned about regular expressions. This might

very well be true, but the main benefit of this book is not in teaching you any practical tool or technique, but instead in giving you a *different way of thinking*: an ability to recognize computational phenomena even when they occur in non-obvious settings, a way to model computational tasks and questions, and to reason about them.

Regardless of any use you will derive from this book, I believe learning this material is important because it contains concepts that are both beautiful and fundamental. The role that *energy* and *matter* played in the 20th century is played in the 21st by *computation* and *information*, not just as tools for our technology and economy, but also as the basic building blocks we use to understand the world. This book will give you a taste of some of the theory behind those, and hopefully spark your curiosity to study more.

0.2 TO POTENTIAL INSTRUCTORS

I wrote this book for my Harvard course, but I hope that other lecturers will find it useful as well. To some extent, it is similar in content to "Theory of Computation" or "Great Ideas" courses such as those taught at CMU or MIT.

The most significant difference between our approach and more traditional ones (such as Hopcroft and Ullman's [HU69; HU79] and Sipser's [Sip97]) is that we do not start with *finite automata* as our initial computational model. Instead, our initial computational model is *Boolean Circuits*.[1] We believe that Boolean Circuits are more fundamental to the theory of computing (and even its practice!) than automata. In particular, Boolean Circuits are a prerequisite for many concepts that one would want to teach in a modern course on theoretical computer science, including cryptography, quantum computing, derandomization, attempts at proving $\mathbf{P} \neq \mathbf{NP}$, and more. Even in cases where Boolean Circuits are not strictly required, they can often offer significant simplifications (as in the case of the proof of the Cook-Levin Theorem).

[1] An earlier book that starts with circuits as the initial model is John Savage's [Sav98].

Furthermore, I believe there are pedagogical reasons to start with Boolean circuits as opposed to finite automata. Boolean circuits are a more natural model of computation, and one that corresponds more closely to computing in silicon, making the connection to practice more immediate to the students. Finite functions are arguably easier to grasp than infinite ones, as we can fully write down their truth table. The theorem that *every* finite function can be computed by some Boolean circuit is both simple enough and important enough to serve as an excellent starting point for this course. Moreover, many of the main conceptual points of the theory of computation, including the notions of the duality between *code* and *data*, and the idea of *universality*, can already be seen in this context.

After Boolean circuits, we move on to Turing machines and prove results such as the existence of a universal Turing machine, the uncomputability of the halting problem, and Rice's Theorem. Automata are discussed after we see Turing machines and undecidability, as an example for a *restricted computational model* where problems such as determining halting can be effectively solved.

While this is not our motivation, the order we present circuits, Turing machines, and automata roughly corresponds to the chronological order of their discovery. Boolean algebra goes back to Boole's and DeMorgan's works in the 1840s [Boo47; De 47] (though the definition of Boolean circuits and the connection to physical computation was given 90 years later by Shannon [Sha38]). Alan Turing defined what we now call "Turing Machines" in the 1930s [Tur37], while finite automata were introduced in the 1943 work of McCulloch and Pitts [MP43] but only really understood in the seminal 1959 work of Rabin and Scott [RS59].

More importantly, while models such as finite-state machines, regular expressions, and context-free grammars are incredibly important for practice, the main applications for these models (whether it is for parsing, for analyzing properties such as *liveness* and *safety*, or even for software-defined routing tables) rely crucially on the fact that these are *tractable* models for which we can effectively answer *semantic questions*. This practical motivation can be better appreciated *after* students see the undecidability of semantic properties of general computing models.

The fact that we start with circuits makes proving the Cook-Levin Theorem much easier. In fact, our proof of this theorem can be (and is) done using a handful of lines of Python. Combining this proof with the standard reductions (which are also implemented in Python) allows students to appreciate visually how a question about computation can be mapped into a question about (for example) the existence of an independent set in a graph.

Some other differences between this book and previous texts are the following:

1. For measuring *time complexity*, we use the standard RAM machine model used (implicitly) in algorithms courses, rather than Turing machines. While these two models are of course polynomially equivalent, and hence make no difference for the definitions of the classes **P**, **NP**, and **EXP**, our choice makes the distinction between notions such as $O(n)$ or $O(n^2)$ time more meaningful. This choice also ensures that these finer-grained time complexity classes correspond to the informal definitions of linear and quadratic time that

students encounter in their algorithms lectures (or their whiteboard coding interviews...).

2. We use the terminology of *functions* rather than *languages*. That is, rather than saying that a Turing Machine M *decides a language* $L \subseteq \{0,1\}^*$, we say that it *computes a function* $F : \{0,1\}^* \to \{0,1\}$. The terminology of "languages" arises from Chomsky's work [Cho56], but it is often more confusing than illuminating. The language terminology also makes it cumbersome to discuss concepts such as algorithms that compute functions with more than one bit of output (including basic tasks such as addition, multiplication, etc...). The fact that we use functions rather than languages means we have to be extra vigilant about students distinguishing between the *specification* of a computational task (e.g., the *function*) and its *implementation* (e.g., the *program*). On the other hand, this point is so important that it is worth repeatedly emphasizing and drilling into the students, regardless of the notation used. The book does mention the language terminology and reminds of it occasionally, to make it easier for students to consult outside resources.

Reducing the time dedicated to finite automata and context-free languages allows instructors to spend more time on topics that a modern course in the theory of computing needs to touch upon. These include randomness and computation, the interactions between *proofs* and *programs* (including Gödel's incompleteness theorem, interactive proof systems, and even a bit on the λ-calculus and the Curry-Howard correspondence), cryptography, and quantum computing.

This book contains sufficient detail to enable its use for self-study. Toward that end, every chapter starts with a list of learning objectives, ends with a recap, and is peppered with "pause boxes" which encourage students to stop and work out an argument or make sure they understand a definition before continuing further.

Section 0.5 contains a "roadmap" for this book, with descriptions of the different chapters, as well as the dependency structure between them. This can help in planning a course based on this book.

0.3 ACKNOWLEDGEMENTS

This text is continually evolving, and I am getting input from many people, for which I am deeply grateful. Salil Vadhan co-taught with me the first iteration of this course and gave me a tremendous amount of useful feedback and insights during this process. Michele Amoretti and Marika Swanberg carefully read several chapters of this text and gave extremely helpful detailed comments. Dave Evans and Richard Xu contributed many pull requests fixing errors and improving phrasing. Thanks to Anil Ada, Venkat Guruswami, and Ryan O'Donnell for

helpful tips from their experience in teaching CMU 15-251. Thanks to Adam Hesterberg and Madhu Sudan for their comments on their experience teaching CS 121 with this book. Kunal Marwaha gave many comments, as well as provided great help with the technical aspects of producing the book.

Thanks to everyone that sent me comments, typo reports, or posted issues or pull requests on the GitHub repository https://github.com/boazbk/tcs. In particular I would like to acknowledge helpful feedback from Scott Aaronson, Michele Amoretti, Aadi Bajpai, Marguerite Basta, Anindya Basu, Sam Benkelman, Jarosław Błasiok, Emily Chan, Christy Cheng, Michelle Chiang, Daniel Chiu, Chi-Ning Chou, Michael Colavita, Brenna Courtney, Rodrigo Daboin Sanchez, Robert Darley Waddilove, Anlan Du, Juan Esteller, David Evans, Michael Fine, Simon Fischer, Leor Fishman, Zaymon Foulds-Cook, William Fu, Kent Furuie, Piotr Galuszka, Carolyn Ge, Jason Giroux, Mark Goldstein, Alexander Golovnev, Sayan Goswami, Maxwell Grozovsky, Michael Haak, Rebecca Hao, Lucia Hoerr, Joosep Hook, Austin Houck, Thomas Huet, Emily Jia, Serdar Kaçka, Chan Kang, Nina Katz-Christy, Vidak Kazic, Joe Kerrigan, Eddie Kohler, Estefania Lahera, Allison Lee, Benjamin Lee, Ondřej Lengál, Raymond Lin, Emma Ling, Alex Lombardi, Lisa Lu, Kai Ma, Aditya Mahadevan, Kunal Marwaha, Christian May, Josh Mehr, Jacob Meyerson, Leon Mlodzian, George Moe, Todd Morrill, Glenn Moss, Haley Mulligan, Hamish Nicholson, Owen Niles, Sandip Nirmel, Sebastian Oberhoff, Thomas Orton, Joshua Pan, Pablo Parrilo, Juan Perdomo, Banks Pickett, Aaron Sachs, Abdelrhman Saleh, Brian Sapozhnikov, Anthony Scemama, Peter Schäfer, Josh Seides, Alaisha Sharma, Nathan Sheely, Haneul Shin, Noah Singer, Matthew Smedberg, Miguel Solano, Hikari Sorensen, David Steurer, Alec Sun, Amol Surati, Everett Sussman, Marika Swanberg, Garrett Tanzer, Eric Thomas, Sarah Turnill, Salil Vadhan, Patrick Watts, Jonah Weissman, Ryan Williams, Licheng Xu, Richard Xu, Wanqian Yang, Elizabeth Yeoh-Wang, Josh Zelinsky, Fred Zhang, Grace Zhang, Alex Zhao, and Jessica Zhu.

I am using many open-source software packages in the production of these notes for which I am grateful. In particular, I am thankful to Donald Knuth and Leslie Lamport for LaTeX and to John MacFarlane for Pandoc. David Steurer wrote the original scripts to produce this text. The current version uses Sergio Correia's panflute. The templates for the LaTeX and HTML versions are derived from Tufte LaTeX, Gitbook and Bookdown. Thanks to Amy Hendrickson for some LaTeX consulting. Juan Esteller and Gabe Montague initially implemented the NAND* programming languages in OCaml and Javascript. I used the Jupyter project to write the supplemental code snippets.

Finally, I would like to thank my family: my wife Ravit, and my children Alma and Goren. Working on this book (and the corresponding course) took so much of my time that Alma wrote an essay for her fifth-grade class saying that "universities should not pressure professors to work too much." I'm afraid all I have to show for this effort is 600 pages of ultra-boring mathematical text.

PRELIMINARIES

Learning Objectives:

- **Introduce and motivate the study of computation for its own sake, irrespective of particular implementations.**
- **The notion of an *algorithm* and some of its history.**
- **Algorithms as not just *tools*, but also *ways of thinking and understanding*.**
- **Taste of Big-O analysis and the surprising creativity in the design of efficient algorithms.**

0 Introduction

"Computer Science is no more about computers than astronomy is about telescopes", attributed to Edsger Dijkstra.[1]

"Hackers need to understand the theory of computation about as much as painters need to understand paint chemistry.", Paul Graham 2003.[2]

"The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why?...I (would like to) show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block.", Alan Cobham, 1964

One of the ancient Babylonians' greatest innovations is the *place-value number system*. The place-value system represents numbers as sequences of digits where the *position* of each digit determines its value.

This is opposed to a system like Roman numerals, where every digit has a fixed value regardless of position. For example, the average distance to the moon is approximately 259,956 Roman miles. In standard Roman numerals, that would be

```
MMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMDCCCCLVI
```

Writing the distance to the *sun* in Roman numerals would require about 100,000 symbols; it would take a 50-page book to contain this single number!

[1] This quote is typically read as disparaging the importance of actual physical computers in Computer Science, but note that telescopes are absolutely essential to astronomy as they provide us with the means to connect theoretical predictions with actual experimental observations.

[2] To be fair, in the following sentence Graham says "you need to know how to calculate time and space complexity and about Turing completeness". This book includes these topics, as well as others such as NP-hardness, randomization, cryptography, quantum computing, and more.

For someone who thinks of numbers in an additive system like Roman numerals, quantities like the distance to the moon or sun are not merely large—they are *unspeakable*: they cannot be expressed or even grasped. It's no wonder that Eratosthenes, the first to calculate the earth's diameter (up to about ten percent error), and Hipparchus, the first to calculate the distance to the moon, used not a Roman-numeral type system but the Babylonian sexagesimal (base 60) place-value system.

0.1 INTEGER MULTIPLICATION: AN EXAMPLE OF AN ALGORITHM

In the language of Computer Science, the place-value system for representing numbers is known as a *data structure*: a set of instructions, or "recipe", for representing objects as symbols. An *algorithm* is a set of instructions, or "recipe", for performing operations on such representations. Data structures and algorithms have enabled amazing applications that have transformed human society, but their importance goes beyond their practical utility. Structures from computer science, such as bits, strings, graphs, and even the notion of a program itself, as well as concepts such as universality and replication, have not just found (many) practical uses but contributed a new language and a new way to view the world.

In addition to coming up with the place-value system, the Babylonians also invented the "standard algorithms" that we were all taught in elementary school for adding and multiplying numbers. These algorithms have been essential throughout the ages for people using abaci, papyrus, or pencil and paper, but in our computer age, do they still serve any purpose beyond torturing third-graders? To see why these algorithms are still very much relevant, let us compare the Babylonian digit-by-digit multiplication algorithm ("grade-school multiplication") with the naive algorithm that multiplies numbers through repeated addition. We start by formally describing both algorithms, see Algorithm 0.1 and Algorithm 0.2.

Algorithm 0.1 — Multiplication via repeated addition.

Input: Non-negative integers x, y

Output: Product $x \cdot y$

1: Let $result \leftarrow 0$.
2: **for** $i = 1, \ldots, y$ **do**
3: $\quad result \leftarrow result + x$
4: **end for**
5: **return** $result$

Algorithm 0.2 — Grade-school multiplication.

Input: Non-negative integers x, y

Output: Product $x \cdot y$

1: Write $x = x_{n-1}x_{n-2}\cdots x_0$ and $y = y_{m-1}y_{m-2}\cdots y_0$ in decimal place-value notation. # *x_0 is the ones digit of x, x_1 is the tens digit, etc.*
2: Let $result \leftarrow 0$
3: **for** $i = 0, \ldots, n-1$ **do**
4: **for** $j = 0, \ldots, m-1$ **do**
5: $result \leftarrow result + 10^{i+j} \cdot x_i \cdot y_j$
6: **end for**
7: **end for**
8: **return** $result$

Both Algorithm 0.1 and Algorithm 0.2 assume that we already know how to add numbers, and Algorithm 0.2 also assumes that we can multiply a number by a power of 10 (which is, after all, a simple shift). Suppose that x and y are two integers of $n = 20$ decimal digits each. (This roughly corresponds to 64 binary digits, which is a common size in many programming languages.) Computing $x \cdot y$ using Algorithm 0.1 entails adding x to itself y times which entails (since y is a 20-digit number) at least 10^{19} additions. In contrast, the grade-school algorithm (i.e., Algorithm 0.2) involves n^2 shifts and single-digit products, and so at most $2n^2 = 800$ single-digit operations. To understand the difference, consider that a grade-schooler can perform a single-digit operation in about 2 seconds, and so would require about $1,600$ seconds (about half an hour) to compute $x \cdot y$ using Algorithm 0.2. In contrast, even though it is more than a billion times faster than a human, if we used Algorithm 0.1 to compute $x \cdot y$ using a modern PC, it would take us $10^{20}/10^9 = 10^{11}$ seconds (which is more than three millennia!) to compute the same result.

Computers have not made algorithms obsolete. On the contrary, the vast increase in our ability to measure, store, and communicate data has led to much higher demand for developing better and more sophisticated algorithms that empower us to make better decisions based on these data. We also see that in no small extent the notion of *algorithm* is independent of the actual computing device that executes it. The digit-by-digit multiplication algorithm is vastly better than iterated addition, regardless of whether the technology we use to implement it is a silicon-based chip, or a third-grader with pen and paper.

Theoretical computer science is concerned with the *inherent* properties of algorithms and computation; namely, those properties that are

independent of current technology. We ask some questions that were already pondered by the Babylonians, such as "what is the best way to multiply two numbers?", but also questions that rely on cutting-edge science such as "could we use the effects of quantum entanglement to factor numbers faster?".

R

Remark 0.3 — Specification, implementation, and analysis of algorithms.. A full description of an algorithm has three components:

- **Specification**: **What** is the task that the algorithm performs (e.g., multiplication in the case of Algorithm 0.1 and Algorithm 0.2.)
- **Implementation**: **How** is the task accomplished: what is the sequence of instructions to be performed. Even though Algorithm 0.1 and Algorithm 0.2 perform the same computational task (i.e., they have the same *specification*), they do it in different ways (i.e., they have different *implementations*).
- **Analysis: Why** does this sequence of instructions achieve the desired task. A full description of Algorithm 0.1 and Algorithm 0.2 will include a *proof* for each one of these algorithms that on input x, y, the algorithm does indeed output $x \cdot y$.

Often as part of the analysis we show that the algorithm is not only **correct** but also **efficient**. That is, we want to show that not only will the algorithm compute the desired task, but will do so in a prescribed number of operations. For example Algorithm 0.2 computes the multiplication function on inputs of n digits using $O(n^2)$ operations, while Algorithm 0.4 (described below) computes the same function using $O(n^{1.6})$ operations. (We define the O notations used here in Section 1.4.8.)

0.2 EXTENDED EXAMPLE: A FASTER WAY TO MULTIPLY (OPTIONAL)

Once you think of the standard digit-by-digit multiplication algorithm, it seems like the "obviously best'' way to multiply numbers. In 1960, the famous mathematician Andrey Kolmogorov organized a seminar at Moscow State University in which he conjectured that every algorithm for multiplying two n digit numbers would require a number of basic operations that is proportional to n^2 ($\Omega(n^2)$ operations, using O-notation as defined in Chapter 1). In other words, Kolmogorov conjectured that in any multiplication algorithm, doubling the number of digits would *quadruple* the number of basic operations

required. A young student named Anatoly Karatsuba was in the audience, and within a week he disproved Kolmogorov's conjecture by discovering an algorithm that requires only about $Cn^{1.6}$ operations for some constant C. Such a number becomes much smaller than n^2 as n grows and so for large n Karatsuba's algorithm is superior to the grade-school one. (For example, Python's implementation switches from the grade-school algorithm to Karatsuba's algorithm for numbers that are 1000 bits or larger.) While the difference between an $O(n^{1.6})$ and an $O(n^2)$ algorithm can be sometimes crucial in practice (see Section 0.3 below), in this book we will mostly ignore such distinctions. However, we describe Karatsuba's algorithm below since it is a good example of how algorithms can often be surprising, as well as a demonstration of the *analysis of algorithms*, which is central to this book and to theoretical computer science at large.

Karatsuba's algorithm is based on a faster way to multiply *two-digit* numbers. Suppose that $x, y \in [100] = \{0, \ldots, 99\}$ are a pair of two-digit numbers. Let's write $\overline{x}$ for the "tens" digit of x, and $\underline{x}$ for the "ones" digit, so that $x = 10\overline{x} + \underline{x}$, and write similarly $y = 10\overline{y} + \underline{y}$ for $\overline{x}, \underline{x}, \overline{y}, \underline{y} \in [10]$. The grade-school algorithm for multiplying x and y is illustrated in Fig. 1.

The grade-school algorithm can be thought of as transforming the task of multiplying a pair of two-digit numbers into *four* single-digit multiplications via the formula

$$(10\overline{x} + \underline{x}) \times (10\overline{y} + \underline{y}) = 100\overline{x}\overline{y} + 10(\overline{x}\underline{y} + \underline{x}\overline{y}) + \underline{x}\underline{y} \qquad (1)$$

Generally, in the grade-school algorithm *doubling* the number of digits in the input results in *quadrupling* the number of operations, leading to an $O(n^2)$ times algorithm. In contrast, Karatsuba's algorithm is based on the observation that we can express Eq. (1) also as

$$(10\overline{x}+\underline{x})\times(10\overline{y}+\underline{y}) = (100-10)\overline{x}\overline{y}+10\left[(\overline{x}+\underline{x})(\overline{y}+\underline{y})\right]-(10-1)\underline{x}\underline{y} \quad (2)$$

which reduces multiplying the two-digit number x and y to computing the following three simpler products: $\overline{x}\overline{y}$, $\underline{x}\underline{y}$ and $(\overline{x}+\underline{x})(\overline{y}+\underline{y})$. By repeating the same strategy recursively, we can reduce the task of multiplying two n-digit numbers to the task of multiplying *three* pairs of $\lfloor n/2 \rfloor + 1$ digit numbers.[3] Since every time we *double* the number of digits we *triple* the number of operations, we will be able to multiply numbers of $n = 2^\ell$ digits using about $3^\ell = n^{\log_2 3} \sim n^{1.585}$ operations.

The above is the intuitive idea behind Karatsuba's algorithm, but is not enough to fully specify it. A complete description of an algorithm entails a *precise specification* of its operations together with its *analysis*:

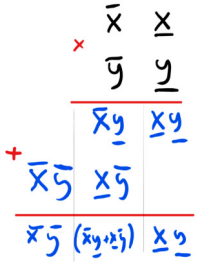


Figure 1: The grade-school multiplication algorithm illustrated for multiplying $x = 10\overline{x} + \underline{x}$ and $y = 10\overline{y} + \underline{y}$. It uses the formula $(10\overline{x} + \underline{x}) \times (10\overline{y} + \underline{y}) = 100\overline{x}\overline{y} + 10(\overline{x}\underline{y} + \underline{x}\overline{y}) + \underline{x}\underline{y}$.

[3] If x is a number then $\lfloor x \rfloor$ is the integer obtained by rounding it down, see Section 1.7.

proof that the algorithm does in fact do what it's supposed to do. The operations of Karatsuba's algorithm are detailed in Algorithm 0.4, while the analysis is given in Lemma 0.5 and Lemma 0.6.

Algorithm 0.4 — Karatsuba multiplication.

Input: non-negative integers x, y each of at most n digits
Output: $x \cdot y$

```
1: procedure KARATSUBA(x,y)
2:     if n ≤ 4 then return x · y ;
3:     Let m = ⌊n/2⌋
4:     Write x = 10^m x̄ + x and y = 10^m ȳ + y
5:     A ← KARATSUBA(x̄, ȳ)
6:     B ← KARATSUBA(x̄ + x, ȳ + y)
7:     C ← KARATSUBA(x, y)
8:     return (10^n − 10^m) · A + 10^m · B + (1 − 10^m) · C
9: end procedure
```

Algorithm 0.4 is only half of the full description of Karatsuba's algorithm. The other half is the *analysis*, which entails proving that **(1)** Algorithm 0.4 indeed computes the multiplication operation and **(2)** it does so using $O(n^{\log_2 3})$ operations. We now turn to showing both facts:

Lemma 0.5 For every non-negative integers x, y, when given input x, y Algorithm 0.4 will output $x \cdot y$.

Proof. Let n be the maximum number of digits of x and y. We prove the lemma by induction on n. The base case is $n \leq 4$ where the algorithm returns $x \cdot y$ by definition. (It does not matter which algorithm we use to multiply four-digit numbers - we can even use repeated addition.) Otherwise, if $n > 4$, we define $m = \lfloor n/2 \rfloor$, and write $x = 10^m \overline{x} + \underline{x}$ and $y = 10^m \overline{y} + \underline{y}$.

Plugging this into $x \cdot y$, we get

$$x \cdot y = 10^{2m} \overline{x}\overline{y} + 10^m (\overline{x}\underline{y} + \underline{x}\overline{y}) + \underline{x}\underline{y} \,. \tag{3}$$

Rearranging the terms we see that

$$x \cdot y = 10^{2m} \overline{x}\overline{y} + 10^m \left[(\overline{x} + \underline{x})(\overline{y} + \underline{y}) - \underline{x}\underline{y} - \overline{x}\overline{y} \right] + \underline{x}\underline{y} \,. \tag{4}$$

since the numbers $\underline{x}, \overline{x}, \underline{y}, \overline{y}, \overline{x} + \underline{x}, \overline{y} + \underline{y}$ all have at most $m + 2 < n$ digits, the induction hypothesis implies that the values A, B, C computed by the recursive calls will satisfy $A = \overline{x}\overline{y}$, $B = (\overline{x} + \underline{x})(\overline{y} + \underline{y})$ and $C = \underline{x}\underline{y}$. Plugging this into (4) we see that $x \cdot y$ equals the value $(10^{2m} - 10^m) \cdot A + 10^m \cdot B + (1 - 10^m) \cdot C$ computed by Algorithm 0.4. ■

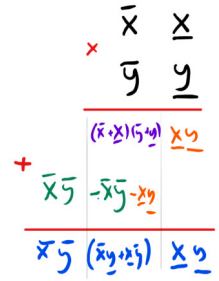


Figure 2: Karatsuba's multiplication algorithm illustrated for multiplying $x = 10\overline{x} + \underline{x}$ and $y = 10\overline{y} + \underline{y}$. We compute the three orange, green and purple products $\underline{x}\underline{y}$, $\overline{x}\overline{y}$ and $(\overline{x} + \underline{x})(\overline{y} + \underline{y})$ and then add and subtract them to obtain the result.

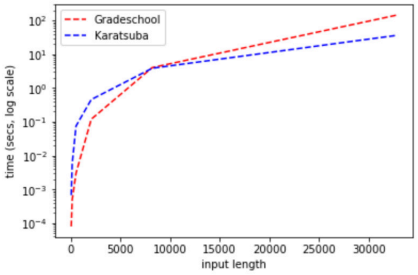


Figure 3: Running time of Karatsuba's algorithm vs. the grade-school algorithm. (Python implementation available online.) Note the existence of a "cutoff" length, where for sufficiently large inputs Karatsuba becomes more efficient than the grade-school algorithm. The precise cutoff location varies by implementation and platform details, but will always occur eventually.

Lemma 0.6 If x, y are integers of at most n digits, Algorithm 0.4 will take $O(n^{\log_2 3})$ operations on input x, y.

Proof. Fig. 2 illustrates the idea behind the proof, which we only sketch here, leaving filling out the details as Exercise 0.4. The proof is again by induction. We define $T(n)$ to be the maximum number of steps that Algorithm 0.4 takes on inputs of length at most n. Since in the base case $n \leq 4$, Exercise 0.4 performs a constant number of computation, we know that $T(4) \leq c$ for some constant c and for $n > 4$, it satisfies the recursive equation

$$T(n) \leq 3T(\lfloor n/2 \rfloor + 1) + c'n \tag{5}$$

for some constant c' (using the fact that addition can be done in $O(n)$ operations).

The recursive equation (5) solves to $O(n^{\log_2 3})$. The intuition behind this is presented in Fig. 2, and this is also a consequence of the so-called "Master Theorem" on recurrence relations. As mentioned above, we leave completing the proof to the reader as Exercise 0.4. ■

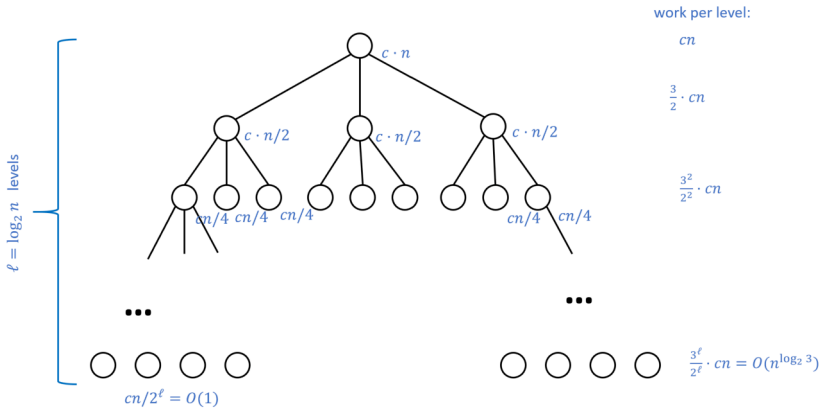


Figure 4: Karatsuba's algorithm reduces an n-bit multiplication to three $n/2$-bit multiplications, which in turn are reduced to nine $n/4$-bit multiplications and so on. We can represent the computational cost of all these multiplications in a 3-ary tree of depth $\log_2 n$, where at the root the extra cost is cn operations, at the first level the extra cost is $c(n/2)$ operations, and at each of the 3^i nodes of level i, the extra cost is $c(n/2^i)$. The total cost is $cn\sum_{i=0}^{\log_2 n}(3/2)^i \leq 10cn^{\log_2 3}$ by the formula for summing a geometric series.

Karatsuba's algorithm is by no means the end of the line for multiplication algorithms. In the 1960's, Toom and Cook extended Karatsuba's ideas to get an $O(n^{\log_k(2k-1)})$ time multiplication algorithm for every constant k. In 1971, Schönhage and Strassen got even better algorithms using the *Fast Fourier Transform*; their idea was to somehow treat integers as "signals" and do the multiplication more efficiently by moving to the Fourier domain. (The *Fourier transform* is a central tool in mathematics and engineering, used in a great many applications; if you have not seen it yet, you are likely to encounter it at some point in your studies.) In the years that followed researchers kept improving the algorithm, and only very recently Harvey and Van Der

Hoeven managed to obtain an $O(n \log n)$ time algorithm for multiplication (though it only starts beating the Schönhage-Strassen algorithm for truly astronomical numbers). Yet, despite all this progress, we still don't know whether or not there is an $O(n)$ time algorithm for multiplying two n digit numbers!

R

Remark 0.7 — Matrix Multiplication (advanced note). (This book contains many "advanced" or "optional" notes and sections. These may assume background that not every student has, and can be safely skipped over as none of the future parts depends on them.)

Ideas similar to Karatsuba's can be used to speed up *matrix* multiplications as well. Matrices are a powerful way to represent linear equations and operations, widely used in numerous applications of scientific computing, graphics, machine learning, and many many more.

One of the basic operations one can do with two matrices is to *multiply* them. For example, if $x = \begin{pmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \end{pmatrix}$ and $y = \begin{pmatrix} y_{0,0} & y_{0,1} \\ y_{1,0} & y_{1,1} \end{pmatrix}$ then the product of x and y is the matrix $\begin{pmatrix} x_{0,0}y_{0,0} + x_{0,1}y_{1,0} & x_{0,0}y_{0,1} + x_{0,1}y_{1,1} \\ x_{1,0}y_{0,0} + x_{1,1}y_{1,0} & x_{1,0}y_{0,1} + x_{1,1}y_{1,1} \end{pmatrix}$. You can see that we can compute this matrix by *eight* products of numbers.

Now suppose that n is even and x and y are a pair of $n \times n$ matrices which we can think of as each composed of four $(n/2) \times (n/2)$ blocks $x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}$ and $y_{0,0}, y_{0,1}, y_{1,0}, y_{1,1}$. Then the formula for the matrix product of x and y can be expressed in the same way as above, just replacing products $x_{a,b}y_{c,d}$ with *matrix* products, and addition with matrix addition. This means that we can use the formula above to give an algorithm that *doubles* the dimension of the matrices at the expense of increasing the number of operations by a factor of 8, which for $n = 2^\ell$ results in $8^\ell = n^3$ operations.

In 1969 Volker Strassen noted that we can compute the product of a pair of two-by-two matrices using only *seven* products of numbers by observing that each entry of the matrix xy can be computed by adding and subtracting the following seven terms: $t_1 = (x_{1,0} + x_{1,1})(y_{0,0} + y_{1,1})$, $t_2 = (x_{0,0} + x_{1,1})y_{0,0}$, $t_3 = x_{0,0}(y_{0,1} - y_{1,1})$, $t_4 = x_{1,1}(y_{0,1} - y_{0,0})$, $t_5 = (x_{0,0} + x_{0,1})y_{1,1}$, $t_6 = (x_{1,0} - x_{0,0})(y_{1,0} + y_{0,1})$, $t_7 = (x_{0,1} - x_{1,1})(y_{1,0} + y_{1,1})$. Indeed, one can verify that $xy = \begin{pmatrix} t_1 + t_4 - t_5 + t_7 & t_3 + t_5 \\ t_2 + t_4 & t_1 + t_3 - t_2 + t_6 \end{pmatrix}$.

Using this observation, we can obtain an algorithm such that doubling the dimension of the matrices results in increasing the number of operations by a factor of 7, which means that for $n = 2^\ell$ the cost is $7^\ell = n^{\log_2 7} \sim n^{2.807}$. A long sequence of work has since improved this algorithm, and the current record has a running time of about $O(n^{2.373})$. However, unlike the case of integer multiplication, at the moment we don't know of any algorithm for matrix multiplication that runs in time linear or even close to linear in the size of the input matrices (e.g., an $O(n^2 polylog(n))$ time algorithm). People have tried to use group representations, which can be thought of as generalizations of the Fourier transform, to obtain faster algorithms, but this effort has not yet succeeded.

0.3 ALGORITHMS BEYOND ARITHMETIC

The quest for better algorithms is by no means restricted to arithmetic tasks such as adding, multiplying or solving equations. Many *graph algorithms*, including algorithms for finding paths, matchings, spanning trees, cuts, and flows, have been discovered in the last several decades, and this is still an intensive area of research. (For example, the last few years saw many advances in algorithms for the *maximum flow* problem, borne out of unexpected connections with electrical circuits and linear equation solvers.) These algorithms are being used not just for the "natural" applications of routing network traffic or GPS-based navigation, but also for applications as varied as drug discovery through searching for structures in gene-interaction graphs to computing risks from correlations in financial investments.

Google was founded based on the *PageRank* algorithm, which is an efficient algorithm to approximate the "principal eigenvector" of (a dampened version of) the adjacency matrix of the web graph. The *Akamai* company was founded based on a new data structure, known as *consistent hashing*, for a hash table where buckets are stored at different servers. The *backpropagation algorithm*, which computes partial derivatives of a neural network in $O(n)$ instead of $O(n^2)$ time, underlies many of the recent phenomenal successes of learning deep neural networks. Algorithms for solving linear equations under sparsity constraints, a concept known as *compressed sensing*, have been used to drastically reduce the amount and quality of data needed to analyze MRI images. This made a critical difference for MRI imaging of cancer tumors in children, where previously doctors needed to use anesthesia to suspend breath during the MRI exam, sometimes with dire consequences.

Even for classical questions, studied through the ages, new discoveries are still being made. For example, for the question of determining whether a given integer is prime or composite, which has been studied since the days of Pythagoras, efficient probabilistic algorithms were only discovered in the 1970s, while the first deterministic polynomial-time algorithm was only found in 2002. For the related problem of actually finding the factors of a composite number, new algorithms were found in the 1980s, and (as we'll see later in this course) discoveries in the 1990s raised the tantalizing prospect of obtaining faster algorithms through the use of quantum mechanical effects.

Despite all this progress, there are still many more questions than answers in the world of algorithms. For almost all natural problems, we do not know whether the current algorithm is the "best", or whether a significantly better one is still waiting to be discovered. As alluded to in Cobham's opening quote for this chapter, even for the basic problem of multiplying numbers we have not yet answered the question of whether there is a multiplication algorithm that is as efficient as our algorithms for addition. But at least we now know the right way to *ask* it.

0.4 ON THE IMPORTANCE OF NEGATIVE RESULTS

Finding better algorithms for problems such as multiplication, solving equations, graph problems, or fitting neural networks to data, is undoubtedly a worthwhile endeavor. But why is it important to prove that such algorithms *don't* exist? One motivation is pure intellectual curiosity. Another reason to study impossibility results is that they correspond to the fundamental limits of our world. In other words, impossibility results are *laws of nature*.

Here are some examples of impossibility results outside computer science (see Section 0.7 for more about these). In physics, the impossibility of building a *perpetual motion machine* corresponds to the *law of conservation of energy*. The impossibility of building a heat engine beating Carnot's bound corresponds to the second law of thermodynamics, while the impossibility of faster-than-light information transmission is a cornerstone of special relativity. In mathematics, while we all learned the formula for solving quadratic equations in high school, the impossibility of generalizing this formula to equations of degree five or more gave birth to *group theory*. The impossibility of proving Euclid's fifth axiom from the first four gave rise to non-Euclidean geometries, which ended up crucial for the theory of general relativity.

In an analogous way, impossibility results for computation correspond to "computational laws of nature" that tell us about the fundamental limits of any information processing apparatus, whether

based on silicon, neurons, or quantum particles. Moreover, computer scientists found creative approaches to *apply* computational limitations to achieve certain useful tasks. For example, much of modern Internet traffic is encrypted using the RSA encryption scheme, the security of which relies on the (conjectured) impossibility of efficiently factoring large integers. More recently, the Bitcoin system uses a digital analog of the "gold standard" where, instead of using a precious metal, new currency is obtained by "mining" solutions for computationally difficult problems.

Chapter Recap

- The history of algorithms goes back thousands of years; they have been essential to much of human progress and these days form the basis of multi-billion dollar industries, as well as life-saving technologies.
- There is often more than one algorithm to achieve the same computational task. Finding a faster algorithm can often make a much bigger difference than improving computing hardware.
- Better algorithms and data structures don't just speed up calculations, but can yield new qualitative insights.
- One question we will study is to find out what is the *most efficient* algorithm for a given problem.
- To show that an algorithm is the most efficient one for a given problem, we need to be able to *prove* that it is *impossible* to solve the problem using a smaller amount of computational resources.

0.5 ROADMAP TO THE REST OF THIS BOOK

Often, when we try to solve a computational problem, whether it is solving a system of linear equations, finding the top eigenvector of a matrix, or trying to rank Internet search results, it is enough to use the "I know it when I see it" standard for describing algorithms. As long as we find some way to solve the problem, we are happy and might not care much on the exact mathematical model for our algorithm. But when we want to answer a question such as "does there *exist* an algorithm to solve the problem P?" we need to be much more precise.

In particular, we will need to **(1)** define exactly what it means to solve P, and **(2)** define exactly what an algorithm is. Even **(1)** can sometimes be non-trivial but **(2)** is particularly challenging; it is not at all clear how (and even whether) we can encompass all potential ways to design algorithms. We will consider several simple *models of computation*, and argue that, despite their simplicity, they do capture

all "reasonable" approaches to achieve computing, including all those that are currently used in modern computing devices.

Once we have these formal models of computation, we can try to obtain *impossibility results* for computational tasks, showing that some problems *can not be solved* (or perhaps can not be solved within the resources of our universe). Archimedes once said that given a fulcrum and a long enough lever, he could move the world. We will see how *reductions* allow us to leverage one hardness result into a slew of a great many others, illuminating the boundaries between the computable and uncomputable (or tractable and intractable) problems.

Later in this book we will go back to examining our models of computation, and see how resources such as randomness or quantum entanglement could potentially change the power of our model. In the context of probabilistic algorithms, we will see a glimpse of how randomness has become an indispensable tool for understanding computation, information, and communication. We will also see how computational difficulty can be an asset rather than a hindrance, and be used for the "derandomization" of probabilistic algorithms. The same ideas also show up in *cryptography*, which has undergone not just a technological but also an intellectual revolution in the last few decades, much of it building on the foundations that we explore in this course.

Theoretical Computer Science is a vast topic, branching out and touching upon many scientific and engineering disciplines. This book provides a very partial (and biased) sample of this area. More than anything, I hope I will manage to "infect" you with at least some of my love for this field, which is inspired and enriched by the connection to practice, but is also deep and beautiful regardless of applications.

0.5.1 Dependencies between chapters

This book is divided into the following parts, see Fig. 5.

- **Preliminaries:** Introduction, mathematical background, and representing objects as strings.
- **Part I: Finite computation (Boolean circuits):** Equivalence of circuits and straight-line programs. Universal gate sets. Existence of a circuit for every function, representing circuits as strings, universal circuit, lower bound on circuit size using the counting argument.
- **Part II: Uniform computation (Turing machines):** Equivalence of Turing machines and programs with loops. Equivalence of models (including RAM machines, λ calculus, and cellular automata), configurations of Turing machines, existence of a universal Turing

machine, uncomputable functions (including the Halting problem and Rice's Theorem), Gödel's incompleteness theorem, restricted computational models (regular and context free languages).

- **Part III: Efficient computation:** Definition of running time, time hierarchy theorem, **P** and **NP**, $\mathbf{P}_{/\mathbf{poly}}$, **NP** completeness and the Cook-Levin Theorem, space bounded computation.
- **Part IV: Randomized computation:** Probability, randomized algorithms, **BPP**, amplification, $\mathbf{BPP} \subseteq \mathbf{P}_{/poly}$, pseudorandom generators and derandomization.
- **Part V: Advanced topics:** Cryptography, proofs and algorithms (interactive and zero knowledge proofs, Curry-Howard correspondence), quantum computing.

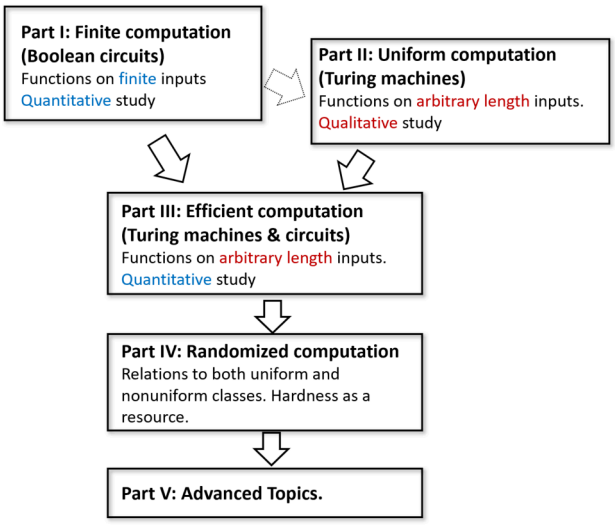


Figure 5: The dependency structure of the different parts. Part I introduces the model of Boolean circuits to study *finite functions* with an emphasis on *quantitative* questions (how many gates to compute a function). Part II introduces the model of Turing machines to study functions that have *unbounded input lengths* with an emphasis on *qualitative* questions (is this function computable or not). Much of Part II does not depend on Part I, as Turing machines can be used as the first computational model. Part III depends on both parts as it introduces a *quantitative* study of functions with unbounded input length. The more advanced parts IV (randomized computation) and V (advanced topics) rely on the material of Parts I, II and III.

The book largely proceeds in linear order, with each chapter building on the previous ones, with the following exceptions:

- The topics of λ calculus (Section 8.5 and Section 8.5), Gödel's incompleteness theorem (Chapter 11), Automata/regular expressions and context-free grammars (Chapter 10), and space-bounded computation (Chapter 17), are not used in the following chapters. Hence you can choose whether to cover or skip any subset of them.
- Part II (Uniform Computation / Turing Machines) does not have a strong dependency on Part I (Finite computation / Boolean circuits) and it should be possible to teach them in the reverse order with minor modification. Boolean circuits are used Part III (efficient computation) for results such as $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$ and the Cook-Levin

Theorem, as well as in Part IV (for $\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$ and derandomization) and Part V (specifically in cryptography and quantum computing).

- All chapters in Part V (Advanced topics) are independent of one another and can be covered in any order.

A course based on this book can use all of Parts I, II, and III (possibly skipping over some or all of the λ calculus, Chapter 11, Chapter 10 or Chapter 17), and then either cover all or some of Part IV (randomized computation), and add a "sprinkling" of advanced topics from Part V based on student or instructor interest.

0.6 EXERCISES

Exercise 0.1 Rank the significance of the following inventions in speeding up the multiplication of large (that is 100-digit or more) numbers. That is, use "back of the envelope" estimates to order them in terms of the speedup factor they offered over the previous state of affairs.

a. Discovery of the grade-school digit by digit algorithm (improving upon repeated addition).

b. Discovery of Karatsuba's algorithm (improving upon the digit by digit algorithm).

c. Invention of modern electronic computers (improving upon calculations with pen and paper).

■

Exercise 0.2 The 1977 Apple II personal computer had a processor speed of 1.023 Mhz or about 10^6 operations per second. At the time of this writing the world's fastest supercomputer performs 93 "petaflops" (10^{15} floating point operations per second) or about 10^{18} basic steps per second. For each one of the following running times (as a function of the input length n), compute for both computers how large an input they could handle in a week of computation, if they run an algorithm that has this running time:

a. n operations.

b. n^2 operations.

c. $n \log n$ operations.

d. 2^n operations.

e. $n!$ operations.

Exercise 0.3 — Usefulness of algorithmic non-existence. In this chapter we mentioned several companies that were founded based on the discovery of new algorithms. Can you give an example for a company that was founded based on the *non-existence* of an algorithm? See footnote for hint.[4]

Exercise 0.4 — Analysis of Karatsuba's Algorithm. a. Suppose that $T_1, T_2, T_3, \ldots$ is a sequence of numbers such that $T_2 \leq 10$ and for every n, $T_n \leq 3T_{\lfloor n/2 \rfloor + 1} + Cn$ for some $C \geq 1$. Prove that $T_n \leq 20Cn^{\log_2 3}$ for every $n > 2$.[5]

b. Prove that the number of single-digit operations that Karatsuba's algorithm takes to multiply two n digit numbers is at most $1000n^{\log_2 3}$.

Exercise 0.5 Implement in the programming language of your choice functions `Gradeschool_multiply(x,y)` and `Karatsuba_multiply(x,y)` that take two arrays of digits `x` and `y` and return an array representing the product of `x` and `y` (where `x` is identified with the number `x[0]+10*x[1]+100*x[2]+...` etc..) using the grade-school algorithm and the Karatsuba algorithm respectively. At what number of digits does the Karatsuba algorithm beat the grade-school one?

Exercise 0.6 — Matrix Multiplication (optional, advanced). In this exercise, we show that if for some $\omega > 2$, we can write the product of two $k \times k$ real-valued matrices A, B using at most k^ω multiplications, then we can multiply two $n \times n$ matrices in roughly n^ω time for every large enough n.

To make this precise, we need to make some notation that is unfortunately somewhat cumbersome. Assume that there is some $k \in \mathbb{N}$ and $m \leq k^\omega$ such that for every $k \times k$ matrices A, B, C such that $C = AB$, we can write for every $i, j \in [k]$:

$$C_{i,j} = \sum_{\ell=0}^{m-1} \alpha_{i,j}^{\ell} f_\ell(A) g_\ell(B)$$

for some linear functions $f_0, \ldots, f_{m-1}, g_0, \ldots, g_{m-1} : \mathbb{R}^{n^2} \to \mathbb{R}$ and coefficients $\{\alpha_{i,j}^{\ell}\}_{i,j \in [k], \ell \in [m]}$. Prove that under this assumption for every $\epsilon > 0$, if n is sufficiently large, then there is an algorithm that computes the product of two $n \times n$ matrices using at most $O(n^{\omega+\epsilon})$ arithmetic operations. See footnote for hint.[6]

[4] As we will see in Chapter Chapter 21, almost any company relying on cryptography needs to assume the *non-existence* of certain algorithms. In particular, RSA Security was founded based on the security of the RSA cryptosystem, which presumes the *non-existence* of an efficient algorithm to compute the prime factorization of large integers.

[5] **Hint:** Use a proof by induction - suppose that this is true for all n's from 1 to m and prove that this is true also for $m + 1$.

[6] Start by showing this for the case that $n = k^t$ for some natural number t, in which case you can do so recursively by breaking the matrices into $k \times k$ blocks.

0.7 BIBLIOGRAPHICAL NOTES

For a brief overview of what we'll see in this book, you could do far worse than read Bernard Chazelle's wonderful essay on the Algorithm as an Idiom of modern science. The book of Moore and Mertens [MM11] gives a wonderful and comprehensive overview of the theory of computation, including much of the content discussed in this chapter and the rest of this book. Aaronson's book [Aar13] is another great read that touches upon many of the same themes.

For more on the algorithms the Babylonians used, see Knuth's paper and Neugebauer's classic book.

Many of the algorithms we mention in this chapter are covered in algorithms textbooks such as those by Cormen, Leiserson, Rivest, and Stein [Cor+09], Kleinberg and Tardos [KT06], and Dasgupta, Papadimitriou and Vazirani [DPV08], as well as Jeff Erickson's textbook. Erickson's book is freely available online and contains a great exposition of recursive algorithms in general and Karatsuba's algorithm in particular.

The story of Karatsuba's discovery of his multiplication algorithm is recounted by him in [Kar95]. As mentioned above, further improvements were made by Toom and Cook [Too63; Coo66], Schönhage and Strassen [SS71], Fürer [Für07], and recently by Harvey and Van Der Hoeven [HV19], see this article for a nice overview. The last papers crucially rely on the *Fast Fourier transform* algorithm. The fascinating story of the (re)discovery of this algorithm by John Tukey in the context of the cold war is recounted in [Coo87]. (We say re-discovery because it later turned out that the algorithm dates back to Gauss [HJB85].) The Fast Fourier Transform is covered in some of the books mentioned below, and there are also online available lectures such as Jeff Erickson's. See also this popular article by David Austin. Fast *matrix* multiplication was discovered by Strassen [Str69], and since then this has been an active area of research. [Blä13] is a recommended self-contained survey of this area.

The *Backpropagation* algorithm for fast differentiation of neural networks was invented by Werbos [Wer74]. The *Pagerank* algorithm was invented by Larry Page and Sergey Brin [Pag+99]. It is closely related to the *HITS* algorithm of Kleinberg [Kle99]. The *Akamai* company was founded based on the *consistent hashing* data structure described in [Kar+97]. *Compressed sensing* has a long history but two foundational papers are [CRT06; Don06]. [Lus+08] gives a survey of applications of compressed sensing to MRI; see also this popular article by Ellenberg [Ell10]. The deterministic polynomial-time algorithm for testing primality was given by Agrawal, Kayal, and Saxena [AKS04].

We alluded briefly to classical impossibility results in mathematics, including the impossibility of proving Euclid's fifth postulate from the other four, impossibility of trisecting an angle with a straightedge and compass and the impossibility of solving a quintic equation via radicals. A geometric proof of the impossibility of angle trisection (one of the three geometric problems of antiquity, going back to the ancient Greeks) is given in this blog post of Tao. The book of Mario Livio [Liv05] covers some of the background and ideas behind these impossibility results. Some exciting recent research is focused on trying to use computational complexity to shed light on fundamental questions in physics such as understanding black holes and reconciling general relativity with quantum mechanics

Learning Objectives:

- Recall basic mathematical notions such as sets, functions, numbers, logical operators and quantifiers, strings, and graphs.
- Rigorously define Big-O notation.
- Proofs by induction.
- Practice with reading mathematical *definitions*, *statements*, and *proofs*.
- Transform an intuitive argument into a rigorous proof.

1
Mathematical Background

> *"I found that every number, which may be expressed from one to ten, surpasses the preceding by one unit: afterwards the ten is doubled or tripled ... until a hundred; then the hundred is doubled and tripled in the same manner as the units and the tens ... and so forth to the utmost limit of numeration."*, Muhammad ibn Mūsā al-Khwārizmī, 820, translation by Fredric Rosen, 1831.

In this chapter we review some of the mathematical concepts that we use in this book. These concepts are typically covered in courses or textbooks on "mathematics for computer science" or "discrete mathematics"; see the "Bibliographical Notes" section (Section 1.9) for several excellent resources on these topics that are freely-available online.

A mathematician's apology. Some students might wonder why this book contains so much math. The reason is that mathematics is simply a language for modeling concepts in a precise and unambiguous way. In this book we use math to model the concept of *computation*. For example, we will consider questions such as *"is there an efficient algorithm to find the prime factors of a given integer?"*. (We will see that this question is particularly interesting, touching on areas as far apart as Internet security and quantum mechanics!) To even *phrase* such a question, we need to give a precise *definition* of the notion of an *algorithm*, and of what it means for an algorithm to be *efficient*. Also, since there is no empirical experiment to prove the *nonexistence* of an algorithm, the only way to establish such a result is using a *mathematical proof*.

1.1 THIS CHAPTER: A READER'S MANUAL

Depending on your background, you can approach this chapter in two different ways:

- If you have already taken "discrete mathematics", "mathematics for computer science" or similar courses, you do not need to read

the whole chapter. You can just take a quick look at Section 1.2 to see the main tools we will use, Section 1.7 for our notation and conventions, and then skip ahead to the rest of this book. Alternatively, you can sit back, relax, and read this chapter just to get familiar with our notation, as well as to enjoy (or not) my philosophical musings and attempts at humor.

- If your background is less extensive, see Section 1.9 for some resources on these topics. This chapter briefly covers the concepts that we need, but you may find it helpful to see a more in-depth treatment. As usual with math, the best way to get comfortable with this material is to work out exercises on your own.
- You might also want to start brushing up on *discrete probability*, which we'll use later in this book (see Chapter 18).

1.2 A QUICK OVERVIEW OF MATHEMATICAL PREREQUISITES

The main mathematical concepts we will use are the following. We just list these notions below, deferring their definitions to the rest of this chapter. If you are familiar with all of these, then you might want to just skip to Section 1.7 to see the full list of notation we use.

- **Proofs:** First and foremost, this book involves a heavy dose of formal mathematical reasoning, which includes mathematical *definitions*, *statements*, and *proofs*.
- **Sets and set operations:** We will use extensively mathematical *sets*. We use the basic set *relations* of membership ($\in$) and containment ($\subseteq$), and set *operations*, principally union ($\cup$), intersection ($\cap$), and set difference ($\setminus$).
- **Cartesian product and Kleene star operation:** We also use the *Cartesian product* of two sets A and B, denoted as $A \times B$ (that is, $A \times B$ the set of pairs (a, b) where $a \in A$ and $b \in B$). We denote by A^n the n fold Cartesian product (e.g., $A^3 = A \times A \times A$) and by A^* (known as the *Kleene star*) the union of A^n for all $n \in \{0, 1, 2, ...\}$.
- **Functions:** The *domain* and *codomain* of a function, properties such as being *one-to-one* (also known as *injective*) or *onto* (also known as *surjective*) functions, as well as *partial functions* (that, unlike standard or "total" functions, are not necessarily defined on all elements of their domain).
- **Logical operations:** The operations AND ($\wedge$), OR ($\vee$), and NOT ($\neg$) and the quantifiers "there exists" ($\exists$) and "for all" ($\forall$).
- **Basic combinatorics:** Notions such as $\binom{n}{k}$ (the number of k-sized subsets of a set of size n).

- **Graphs:** Undirected and directed graphs, connectivity, paths, and cycles.

- **Big-O notation:** $O, o, \Omega, \omega, \Theta$ notation for analyzing asymptotic growth of functions.

- **Discrete probability:** We will use *probability theory*, and specifically probability over *finite* samples spaces such as tossing n coins, including notions such as *random variables*, *expectation*, and *concentration*. We will only use probability theory in the second half of this text, and will review it beforehand in Chapter 18. However, probabilistic reasoning is a subtle (and extremely useful!) skill, and it's always good to start early in acquiring it.

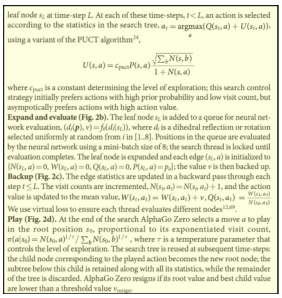
In the rest of this chapter we briefly review the above notions. This is partially to remind the reader and reinforce material that might not be fresh in your mind, and partially to introduce our notation and conventions which might occasionally differ from those you've encountered before.

1.3 READING MATHEMATICAL TEXTS

Mathematicians use jargon for the same reason that it is used in many other professions such as engineering, law, medicine, and others. We want to make terms *precise* and introduce shorthand for concepts that are frequently reused. Mathematical texts tend to "pack a lot of punch" per sentence, and so the key is to read them slowly and carefully, parsing each symbol at a time.

With time and practice you will see that reading mathematical texts becomes easier and jargon is no longer an issue. Moreover, reading mathematical texts is one of the most transferable skills you could take from this book. Our world is changing rapidly, not just in the realm of technology, but also in many other human endeavors, whether it is medicine, economics, law or even culture. Whatever your future aspirations, it is likely that you will encounter texts that use new concepts that you have not seen before (see Fig. 1.1 and Fig. 1.2 for two recent examples from current "hot areas"). Being able to internalize and then apply new definitions can be hugely important. It is a skill that's much easier to acquire in the relatively safe and stable context of a mathematical course, where one at least has the guarantee that the concepts are fully specified, and you have access to your teaching staff for questions.

The basic components of a mathematical text are **definitions**, **assertions** and **proofs**.



leaf node s_l at time-step L. At each of these time-steps, $t<L$, an action is selected according to the statistics in the search tree, $a_t = \text{argmax}_a(Q(s_t,a)+U(s_t,a))$, using a variant of the PUCT algorithm[24],

$$U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

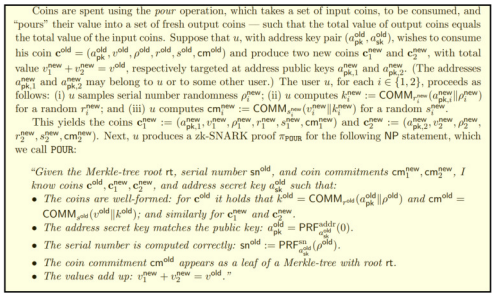
where c_{puct} is a constant determining the level of exploration; this search control strategy initially prefers actions with high prior probability and low visit count, but asymptotically prefers actions with high action value.

Expand and evaluate (Fig. 2b). The leaf node s_l is added to a queue for neural network evaluation, $(d_i(\boldsymbol{p}), v) = f_i(d_i(s_l))$, where d_i is a dihedral reflection or rotation selected uniformly at random from i in $[1..8]$. Positions in the queue are evaluated by the neural network using a mini-batch size of 8; the search thread is locked until evaluation completes. The leaf node is expanded and each edge (s_l, a) is initialized to $\{N(s_l,a)=0, W(s_l,a)=0, Q(s_l,a)=0, P(s_l,a)=p_a\}$; the value v is then backed up.

Backup (Fig. 2c). The edge statistics are updated in a backward pass through each step $t \le L$. The visit counts are incremented, $N(s_t,a_t) = N(s_t,a_t)+1$, and the action value is updated to the mean value, $W(s_t,a_t) = W(s_t,a_t)+v$, $Q(s_t,a_t) = \frac{W(s_t,a_t)}{N(s_t,a_t)}$. We use virtual loss to ensure each thread evaluates different nodes[12,69].

Play (Fig. 2d). At the end of the search AlphaGo Zero selects a move a to play in the root position s_0, proportional to its exponentiated visit count, $\pi(a|s_0) = N(s_0,a)^{1/\tau} / \sum_b N(s_0,b)^{1/\tau}$, where τ is a temperature parameter that controls the level of exploration. The search tree is reused at subsequent time-steps: the child node corresponding to the played action becomes the new root node; the subtree below this child is retained along with all its statistics, while the remainder of the tree is discarded. AlphaGo Zero resigns if its root value and best child value are lower than a threshold value v_{resign}.

Figure 1.1: A snippet from the "methods" section of the "AlphaGo Zero" paper by Silver et al, *Nature*, 2017.



Coins are spent using the *pour* operation, which takes a set of input coins, to be consumed, and "pours" their value into a set of fresh output coins — such that the total value of output coins equals the total value of the input coins. Suppose that u, with address key pair $(a_{\mathsf{pk}}^{\mathsf{old}}, a_{\mathsf{sk}}^{\mathsf{old}})$, wishes to consume his coin $\mathbf{c}^{\mathsf{old}} = (a_{\mathsf{pk}}^{\mathsf{old}}, v^{\mathsf{old}}, \rho^{\mathsf{old}}, r^{\mathsf{old}}, s^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}})$ and produce two new coins $\mathbf{c}_1^{\mathsf{new}}$ and $\mathbf{c}_2^{\mathsf{new}}$, with total value $v_1^{\mathsf{new}} + v_2^{\mathsf{new}} = v^{\mathsf{old}}$, respectively targeted at address public keys $a_{\mathsf{pk},1}^{\mathsf{new}}$ and $a_{\mathsf{pk},2}^{\mathsf{new}}$. (The addresses $a_{\mathsf{pk},1}^{\mathsf{new}}$ and $a_{\mathsf{pk},2}^{\mathsf{new}}$ may belong to u or to some other user.) The user u, for each $i \in \{1,2\}$, proceeds as follows: (i) u samples serial number randomness ρ_i^{new}; (ii) u computes $k_i^{\mathsf{new}} := \mathsf{COMM}_r^{\rho_i^{\mathsf{new}}}(a_{\mathsf{pk},i}^{\mathsf{new}} \| \rho_i^{\mathsf{new}})$ for a random r_i^{new}; and (iii) u computes $\mathsf{cm}_i^{\mathsf{new}} := \mathsf{COMM}_s^{r_i^{\mathsf{new}}}(v_i^{\mathsf{new}} \| k_i^{\mathsf{new}})$ for a random s_i^{new}.

This yields the coins $\mathbf{c}_1^{\mathsf{new}} := (a_{\mathsf{pk},1}^{\mathsf{new}}, v_1^{\mathsf{new}}, \rho_1^{\mathsf{new}}, r_1^{\mathsf{new}}, s_1^{\mathsf{new}}, \mathsf{cm}_1^{\mathsf{new}})$ and $\mathbf{c}_2^{\mathsf{new}} := (a_{\mathsf{pk},2}^{\mathsf{new}}, v_2^{\mathsf{new}}, \rho_2^{\mathsf{new}}, r_2^{\mathsf{new}}, s_2^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}})$. Next, u produces a zk-SNARK proof π_{POUR} for the following NP statement, which we call POUR:

"*Given the Merkle-tree root* rt*, serial number* $\mathsf{sn}^{\mathsf{old}}$*, and coin commitments* $\mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}$*, I know coins* $\mathbf{c}^{\mathsf{old}}, \mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$*, and address secret key* $a_{\mathsf{sk}}^{\mathsf{old}}$ *such that:*

- *The coins are well-formed: for* $\mathbf{c}^{\mathsf{old}}$ *it holds that* $k^{\mathsf{old}} = \mathsf{COMM}_{\rho^{\mathsf{old}}}(a_{\mathsf{pk}}^{\mathsf{old}} \| \rho^{\mathsf{old}})$ *and* $\mathsf{cm}^{\mathsf{old}} = \mathsf{COMM}_{r^{\mathsf{old}}}(v^{\mathsf{old}} \| k^{\mathsf{old}})$*; and similarly for* $\mathbf{c}_1^{\mathsf{new}}$ *and* $\mathbf{c}_2^{\mathsf{new}}$*.*
- *The address secret key matches the public key:* $a_{\mathsf{pk}}^{\mathsf{old}} = \mathsf{PRF}_0^{\mathsf{addr}}(a_{\mathsf{sk}}^{\mathsf{old}})$*.*
- *The serial number is computed correctly:* $\mathsf{sn}^{\mathsf{old}} := \mathsf{PRF}_{a_{\mathsf{sk}}^{\mathsf{old}}}^{\mathsf{sn}}(\rho^{\mathsf{old}})$*.*
- *The coin commitment* $\mathsf{cm}^{\mathsf{old}}$ *appears as a leaf of a Merkle-tree with root* rt*.*
- *The values add up:* $v_1^{\mathsf{new}} + v_2^{\mathsf{new}} = v^{\mathsf{old}}$*."*

Figure 1.2: A snippet from the "Zerocash" paper of Ben-Sasson et al, that forms the basis of the cryptocurrency startup Zcash.

1.3.1 Definitions

Mathematicians often define new concepts in terms of old concepts. For example, here is a mathematical definition which you may have encountered in the past (and will see again shortly):

Definition 1.1 — One to one function. Let S, T be sets. We say that a function $f : S \to T$ is *one to one* (also known as *injective*) if for every two elements $x, x' \in S$, if $x \neq x'$ then $f(x) \neq f(x')$.

Definition 1.1 captures a simple concept, but even so it uses quite a bit of notation. When reading such a definition, it is often useful to annotate it with a pen as you're going through it (see Fig. 1.3). For example, when you see an identifier such as f, S or x, make sure that you realize what sort of object it is: is it a set, a function, an element, a number, a gremlin? You might also find it useful to explain the definition in words to a friend (or to yourself).

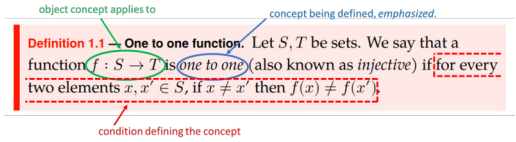


Figure 1.3: An annotated form of Definition 1.1, marking which part is being defined and how.

1.3.2 Assertions: Theorems, lemmas, claims

Theorems, lemmas, claims and the like are true statements about the concepts we defined. Deciding whether to call a particular statement a "Theorem", a "Lemma" or a "Claim" is a judgement call, and does not make a mathematical difference. All three correspond to statements which were proven to be true. The difference is that a *Theorem* refers to a significant result that we would want to remember and highlight. A *Lemma* often refers to a technical result that is not necessarily important in its own right, but that can be often very useful in proving other theorems. A *Claim* is a "throwaway" statement that we need to use in order to prove some other bigger results, but do not care so much about for its own sake.

1.3.3 Proofs

Mathematical *proofs* are the arguments we use to demonstrate that our theorems, lemmas, and claims are indeed true. We discuss proofs in Section 1.5 below, but the main point is that the mathematical standard of proof is very high. Unlike in some other realms, in mathematics a proof is an "airtight" argument that demonstrates that the statement is true beyond a shadow of a doubt. Some examples in this section for mathematical proofs are given in Solved Exercise 1.1 and Section 1.6. As mentioned in the preface, as a general rule, it is more important you understand the **definitions** than the **theorems**, and it is more important you understand a **theorem statement** than its **proof**.

1.4 BASIC DISCRETE MATH OBJECTS

In this section we quickly review some of the mathematical objects (the "basic data structures" of mathematics, if you will) we use in this book.

1.4.1 Sets

A *set* is an unordered collection of objects. For example, when we write $S = \{2, 4, 7\}$, we mean that S denotes the set that contains the numbers 2, 4, and 7. (We use the notation "$2 \in S$" to denote that 2 is an element of S.) Note that the set $\{2, 4, 7\}$ and $\{7, 4, 2\}$ are identical, since they contain the same elements. Also, a set either contains an element or does not contain it – there is no notion of containing it "twice" – and so we could even write the same set S as $\{2, 2, 4, 7\}$ (though that would be a little weird). The *cardinality* of a finite set S, denoted by $|S|$, is the number of elements it contains. (Cardinality can be defined for *infinite* sets as well; see the sources in Section 1.9.) So, in the example above, $|S| = 3$. A set S is a *subset* of a set T, denoted by $S \subseteq T$, if every element of S is also an element of T. (We can also describe this by saying that T is a *superset* of S.) For example, $\{2, 7\} \subseteq \{2, 4, 7\}$. The set that contains no elements is known as the *empty set* and it is denoted by $\emptyset$. If A is a subset of B that is not equal to B we say that A is a *strict subset* of B, and denote this by $A \subsetneq B$.

We can define sets by either listing all their elements or by writing down a rule that they satisfy such as

$$\text{EVEN} = \{x \mid \ x = 2y \text{ for some non-negative integer } y\} \ .$$

Of course there is more than one way to write the same set, and often we will use intuitive notation listing a few examples that illustrate the rule. For example, we can also define EVEN as

$$\text{EVEN} = \{0, 2, 4, ...\} \ .$$

Note that a set can be either finite (such as the set $\{2, 4, 7\}$) or infinite (such as the set EVEN). Also, the elements of a set don't have to be numbers. We can talk about the sets such as the set $\{a, e, i, o, u\}$ of all the vowels in the English language, or the set {`New York`, `Los Angeles`, `Chicago`, `Houston`, `Philadelphia`, `Phoenix`, `San Antonio`, `San Diego`, `Dallas`} of all cities in the U.S. with population more than one million per the 2010 census. A set can even have other sets as elements, such as the set $\{\emptyset, \{1, 2\}, \{2, 3\}, \{1, 3\}\}$ of all even-sized subsets of $\{1, 2, 3\}$.

Operations on sets: The *union* of two sets S, T, denoted by $S \cup T$, is the set that contains all elements that are either in S *or* in T. The *intersection* of S and T, denoted by $S \cap T$, is the set of elements that are

both in S *and* in T. The *set difference* of S and T, denoted by $S \setminus T$ (and in some texts also by $S - T$), is the set of elements that are in S but *not* in T.

Tuples, lists, strings, sequences: A *tuple* is an *ordered* collection of items. For example $(1, 5, 2, 1)$ is a tuple with four elements (also known as a 4-tuple or quadruple). Since order matters, this is not the same tuple as the 4-tuple $(1, 1, 5, 2)$ or the 3-tuple $(1, 5, 2)$. A 2-tuple is also known as a *pair*. We use the terms *tuples* and *lists* interchangeably. A tuple where every element comes from some finite set Σ (such as $\{0, 1\}$) is also known as a *string*. Analogously to sets, we denote the *length* of a tuple T by $|T|$. Just like sets, we can also think of infinite analogues of tuples, such as the ordered collection $(1, 4, 9, ...)$ of all perfect squares. Infinite ordered collections are known as *sequences*; we might sometimes use the term "infinite sequence" to emphasize this, and use "finite sequence" as a synonym for a tuple. (We can identify a sequence $(a_0, a_1, a_2, ...)$ of elements in some set S with a *function* $A : \mathbb{N} \to S$ (where $a_n = A(n)$ for every $n \in \mathbb{N}$). Similarly, we can identify a k-tuple $(a_0, ..., a_{k-1})$ of elements in S with a function $A : [k] \to S$.)

Cartesian product: If S and T are sets, then their *Cartesian product*, denoted by $S \times T$, is the set of all ordered pairs (s, t) where $s \in S$ and $t \in T$. For example, if $S = \{1, 2, 3\}$ and $T = \{10, 12\}$, then $S \times T$ contains the 6 elements $(1, 10), (2, 10), (3, 10), (1, 12), (2, 12), (3, 12)$. Similarly if S, T, U are sets then $S \times T \times U$ is the set of all ordered triples (s, t, u) where $s \in S$, $t \in T$, and $u \in U$. More generally, for every positive integer n and sets $S_0, ..., S_{n-1}$, we denote by $S_0 \times S_1 \times \cdots \times S_{n-1}$ the set of ordered n-tuples $(s_0, ..., s_{n-1})$ where $s_i \in S_i$ for every $i \in \{0, ..., n-1\}$. For every set S, we denote the set $S \times S$ by S^2, $S \times S \times S$ by S^3, $S \times S \times S \times S$ by S^4, and so on and so forth.

1.4.2 Special sets

There are several sets that we will use in this book time and again. The set

$$\mathbb{N} = \{0, 1, 2, ...\}$$

contains all *natural numbers*, i.e., non-negative integers. For any natural number $n \in \mathbb{N}$, we define the set $[n]$ as $\{0, ..., n-1\} = \{k \in \mathbb{N} : k < n\}$. (We start our indexing of both $\mathbb{N}$ and $[n]$ from 0, while many other texts index those sets from 1. Starting from zero or one is simply a convention that doesn't make much difference, as long as one is consistent about it.)

We will also occasionally use the set $\mathbb{Z} = \{..., -2, -1, 0, +1, +2, ...\}$ of (negative and non-negative) *integers*,[1] as well as the set $\mathbb{R}$ of *real*

[1] The letter Z stands for the German word "Zahlen", which means *numbers*.

numbers. (This is the set that includes not just the integers, but also fractional and irrational numbers; e.g., $\mathbb{R}$ contains numbers such as $+0.5$, $-\pi$, etc.) We denote by $\mathbb{R}_+$ the set $\{x \in \mathbb{R} : x > 0\}$ of *positive* real numbers. This set is sometimes also denoted as $(0, \infty)$.

Strings: Another set we will use time and again is

$$\{0,1\}^n = \{(x_0, \ldots, x_{n-1}) \ : \ x_0, \ldots, x_{n-1} \in \{0,1\}\}$$

which is the set of all n-length binary strings for some natural number n. That is $\{0,1\}^n$ is the set of all n-tuples of zeroes and ones. This is consistent with our notation above: $\{0,1\}^2$ is the Cartesian product $\{0,1\} \times \{0,1\}$, $\{0,1\}^3$ is the product $\{0,1\} \times \{0,1\} \times \{0,1\}$ and so on.

We will write the string $(x_0, x_1, \ldots, x_{n-1})$ as simply $x_0 x_1 \cdots x_{n-1}$. For example,

$$\{0,1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\} \ .$$

For every string $x \in \{0,1\}^n$ and $i \in [n]$, we write x_i for the ith element of x.

We will also often talk about the set of binary strings of *all* lengths, which is

$$\{0,1\}^* = \{(x_0, \ldots, x_{n-1}) \ : \ n \in \mathbb{N} \ , \ , x_0, \ldots, x_{n-1} \in \{0,1\}\} \ .$$

Another way to write this set is as

$$\{0,1\}^* = \{0,1\}^0 \cup \{0,1\}^1 \cup \{0,1\}^2 \cup \cdots$$

or more concisely as

$$\{0,1\}^* = \cup_{n \in \mathbb{N}} \{0,1\}^n \ .$$

The set $\{0,1\}^*$ includes the "string of length 0" or "the empty string", which we will denote by "". (In using this notation we follow the convention of many programming languages. Other texts sometimes use ϵ or λ to denote the empty string.)

Generalizing the star operation: For every set Σ, we define

$$\Sigma^* = \cup_{n \in \mathbb{N}} \Sigma^n \ .$$

For example, if $\Sigma = \{a, b, c, d, \ldots, z\}$ then Σ^* denotes the set of all finite length strings over the alphabet a-z.

Concatenation: The *concatenation* of two strings $x \in \Sigma^n$ and $y \in \Sigma^m$ is the $(n+m)$-length string xy obtained by writing y after x. That is, if $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$, then xy is equal to the string $z \in \{0,1\}^{n+m}$ such that for $i \in [n]$, $z_i = x_i$ and for $i \in \{n, \ldots, n+m-1\}$, $z_i = y_{i-n}$.

1.4.3 Functions

If S and T are non-empty sets, a *function* F mapping S to T, denoted by $F : S \to T$, associates with every element $x \in S$ an element $F(x) \in T$. The set S is known as the *domain* of F and the set T is known as the *codomain* of F. The *image* of a function F is the set $\{F(x) \mid x \in S\}$ which is the subset of F's codomain consisting of all output elements that are mapped from some input. (Some texts use *range* to denote the image of a function, while other texts use *range* to denote the codomain of a function. Hence we will avoid using the term "range" altogether.) As in the case of sets, we can write a function either by listing the table of all the values it gives for elements in S or by using a rule. For example if $S = \{0,1,2,3,4,5,6,7,8,9\}$ and $T = \{0,1\}$, then the table below defines a function $F : S \to T$. Note that this function is the same as the function defined by the rule $F(x) = (x \mod 2)$.[2]

[2] For two natural numbers x and a, $x \mod a$ (shorthand for "modulo") denotes the *remainder* of x when it is divided by a. That is, it is the number r in $\{0,\ldots,a-1\}$ such that $x = ak + r$ for some integer k. We sometimes also use the notation $x = y \pmod a$ to denote the assertion that $x \mod a$ is the same as $y \mod a$.

Table 1.1: An example of a function.

Input	Output
0	0
1	1
2	0
3	1
4	0
5	1
6	0
7	1
8	0
9	1

If $F : S \to T$ satisfies that $F(x) \neq F(y)$ for all $x \neq y$ then we say that F is *one-to-one* (Definition 1.1, also known as an *injective* function or simply an *injection*). If F satisfies that for every $y \in T$ there is some $x \in S$ such that $F(x) = y$ then we say that F is *onto* (also known as a *surjective* function or simply a *surjection*). A function that is both one-to-one and onto is known as a *bijective* function or simply a *bijection*. A bijection from a set S to itself is also known as a *permutation* of S. If $F : S \to T$ is a bijection then for every $y \in T$ there is a unique $x \in S$ such that $F(x) = y$. We denote this value x by $F^{-1}(y)$. Note that F^{-1} is itself a bijection from T to S (can you see why?).

Giving a bijection between two sets is often a good way to show they have the same size. In fact, the standard mathematical definition of the notion that "S and T have the same cardinality" is that there

exists a bijection $f : S \to T$. Further, the cardinality of a set S is defined to be n if there is a bijection from S to the set $\{0, \ldots, n-1\}$. As we will see later in this book, this is a definition that generalizes to defining the cardinality of *infinite* sets.

Partial functions: We will sometimes be interested in *partial* functions from S to T. A partial function is allowed to be undefined on some subset of S. That is, if F is a partial function from S to T, then for every $s \in S$, either there is (as in the case of standard functions) an element $F(s)$ in T, or $F(s)$ is undefined. For example, the partial function $F(x) = \sqrt{x}$ is only defined on non-negative real numbers. When we want to distinguish between partial functions and standard (i.e., non-partial) functions, we will call the latter *total* functions. When we say "function" without any qualifier then we mean a *total* function.

The notion of partial functions is a strict generalization of functions, and so every function is a partial function, but not every partial function is a function. (That is, for every non-empty S and T, the set of partial functions from S to T is a proper superset of the set of total functions from S to T.) When we want to emphasize that a function f from A to B might not be total, we will write $f : A \to_p B$. We can think of a partial function F from S to T also as a total function from S to $T \cup \{\perp\}$ where $\perp$ is a special "failure symbol". So, instead of saying that F is undefined at x, we can say that $F(x) = \perp$.

Basic facts about functions: Verifying that you can prove the following results is an excellent way to brush up on functions:

- If $F : S \to T$ and $G : T \to U$ are one-to-one functions, then their *composition* $H : S \to U$ defined as $H(s) = G(F(s))$ is also one to one.
- If $F : S \to T$ is one to one, then there exists an onto function $G : T \to S$ such that $G(F(s)) = s$ for every $s \in S$.
- If $G : T \to S$ is onto then there exists a one-to-one function $F : S \to T$ such that $G(F(s)) = s$ for every $s \in S$.
- If S and T are non-empty finite sets then the following conditions are equivalent to one another: **(a)** $|S| \leq |T|$, **(b)** there is a one-to-one function $F : S \to T$, and **(c)** there is an onto function $G : T \to S$. These equivalences are in fact true even for *infinite* S and T. For infinite sets the condition **(b)** (or equivalently, **(c)**) is the commonly accepted *definition* for $|S| \leq |T|$.

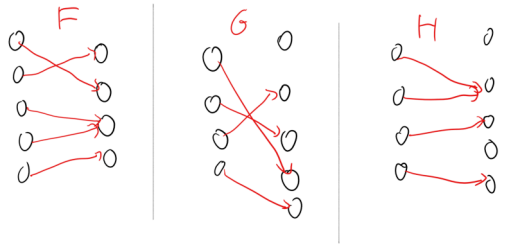


Figure 1.4: We can represent finite functions as a directed graph where we put an edge from x to $f(x)$. The *onto* condition corresponds to requiring that every vertex in the codomain of the function has in-degree *at least* one. The *one-to-one* condition corresponds to requiring that every vertex in the codomain of the function has in-degree *at most* one. In the examples above F is an onto function, G is one to one, and H is neither onto nor one to one.

You can find the proofs of these results in many discrete math texts, including for example, Section 4.5 in the Lehman-Leighton-Meyer notes. However, I strongly suggest you try to prove them on your own, or at least convince yourself that they are true by proving special cases of those for small sizes (e.g., $|S| = 3, |T| = 4, |U| = 5$).

Let us prove one of these facts as an example:

Lemma 1.2 If S, T are non-empty sets and $F : S \to T$ is one to one, then there exists an onto function $G : T \to S$ such that $G(F(s)) = s$ for every $s \in S$.

Proof. Choose some $s_0 \in S$. We will define the function $G : T \to S$ as follows: for every $t \in T$, if there is some $s \in S$ such that $F(s) = t$ then set $G(t) = s$ (the choice of s is well defined since by the one-to-one property of F, there cannot be two distinct s, s' that both map to t). Otherwise, set $G(t) = s_0$. Now for every $s \in S$, by the definition of G, if $t = F(s)$ then $G(t) = G(F(s)) = s$. Moreover, this also shows that G is *onto*, since it means that for every $s \in S$ there is some t, namely $t = F(s)$, such that $G(t) = s$. ■

1.4.4 Graphs

Graphs are ubiquitous in Computer Science, and many other fields as well. They are used to model a variety of data types including social networks, scheduling constraints, road networks, deep neural nets, gene interactions, correlations between observations, and a great many more. Formal definitions of several kinds of graphs are given next, but if you have not seen graphs before in a course, I urge you to read up on them in one of the sources mentioned in Section 1.9.

Graphs come in two basic flavors: *undirected* and *directed*.[3]

Definition 1.3 — Undirected graphs. An *undirected graph* $G = (V, E)$ consists of a set V of *vertices* and a set E of edges. Every edge is a size two subset of V. We say that two vertices $u, v \in V$ are *neighbors*, if the edge $\{u, v\}$ is in E.

Given this definition, we can define several other properties of graphs and their vertices. We define the *degree* of u to be the number of neighbors u has. A *path* in the graph is a tuple $(u_0, \ldots, u_k) \in V^{k+1}$, for some $k > 0$ such that u_{i+1} is a neighbor of u_i for every $i \in [k]$. A *simple path* is a path $(u_0, \ldots, u_{k-1})$ where all the u_i's are distinct. A *cycle* is a path $(u_0, \ldots, u_k)$ where $u_0 = u_k$. We say that two vertices $u, v \in V$ are *connected* if either $u = v$ or there is a path from $(u_0, \ldots, u_k)$ where

[3] It is possible, and sometimes useful, to think of an undirected graph as the special case of a directed graph that has the special property that for every pair u, v either both the edges (u, v) and (v, u) are present or neither of them is. However, in many settings there is a significant difference between undirected and directed graphs, and so it's typically best to think of them as separate categories.

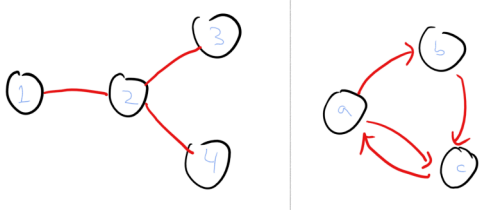


Figure 1.5: An example of an undirected and a directed graph. The undirected graph has vertex set $\{1, 2, 3, 4\}$ and edge set $\{\{1, 2\}, \{2, 3\}, \{2, 4\}\}$. The directed graph has vertex set $\{a, b, c\}$ and the edge set $\{(a, b), (b, c), (c, a), (a, c)\}$.

$u_0 = u$ and $u_k = v$. We say that the graph G is *connected* if every pair of vertices in it is connected.

Here are some basic facts about undirected graphs. We give some informal arguments below, but leave the full proofs as exercises (the proofs can be found in many of the resources listed in Section 1.9).

Lemma 1.4 In any undirected graph $G = (V, E)$, the sum of the degrees of all vertices is equal to twice the number of edges.

Lemma 1.4 can be shown by seeing that every edge $\{u, v\}$ contributes twice to the sum of the degrees (once for u and the second time for v).

Lemma 1.5 The connectivity relation is *transitive*, in the sense that if u is connected to v, and v is connected to w, then u is connected to w.

Lemma 1.5 can be shown by simply attaching a path of the form $(u, u_1, u_2, \ldots, u_{k-1}, v)$ to a path of the form $(v, u'_1, \ldots, u'_{k'-1}, w)$ to obtain the path $(u, u_1, \ldots, u_{k-1}, v, u'_1, \ldots, u'_{k'-1}, w)$ that connects u to w.

Lemma 1.6 For every undirected graph $G = (V, E)$ and connected pair u, v, the shortest path from u to v is simple. In particular, for every connected pair there exists a simple path that connects them.

Lemma 1.6 can be shown by "shortcutting" any non-simple path from u to v where the same vertex w appears twice to remove it (see Fig. 1.6). It is a good exercise to transforming this intuitive reasoning to a formal proof:

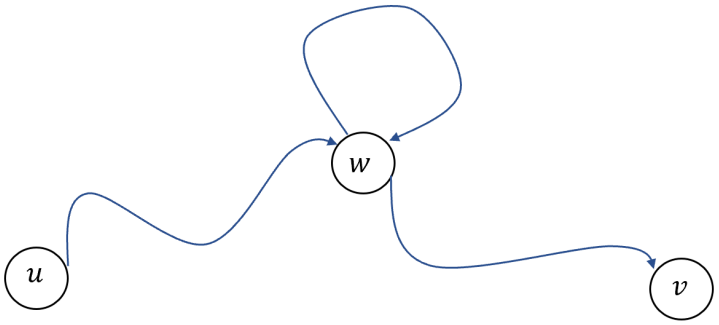


Figure 1.6: If there is a path from u to v in a graph that passes twice through a vertex w then we can "shortcut" it by removing the loop from w to itself to find a path from u to v that only passes once through w.

Solved Exercise 1.1 — Connected vertices have simple paths. Prove Lemma 1.6 ■

Solution:

The proof follows the idea illustrated in Fig. 1.6. One complication is that there can be more than one vertex that is visited twice

by a path, and so "shortcutting" might not necessarily result in a simple path; we deal with this by looking at a *shortest* path between u and v. Details follow.

Let $G = (V, E)$ be a graph and u and v in V be two connected vertices in G. We will prove that there is a simple path between u and v. Let k be the shortest length of a path between u and v and let $P = (u_0, u_1, u_2, \ldots, u_{k-1}, u_k)$ be a k-length path from u to v (there can be more than one such path: if so we just choose one of them). (That is $u_0 = u$, $u_k = v$, and $(u_\ell, u_{\ell+1}) \in E$ for all $\ell \in [k]$.) We claim that P is simple. Indeed, suppose otherwise that there is some vertex w that occurs twice in the path: $w = u_i$ and $w = u_j$ for some $i < j$. Then we can "shortcut" the path P by considering the path $P' = (u_0, u_1, \ldots, u_{i-1}, w, u_{j+1}, \ldots, u_k)$ obtained by taking the first i vertices of P (from $u_0 = 0$ to the first occurrence of w) and the last $k - j$ ones (from the vertex u_{j+1} following the second occurrence of w to $u_k = v$). The path P' is a valid path between u and v since every consecutive pair of vertices in it is connected by an edge (in particular, since $w = u_i = u_j$, both (u_{i-1}, w) and (w, u_{j+1}) are edges in E), but since the length of P' is $k - (j - i) < k$, this contradicts the minimality of P.

■

R

Remark 1.7 — Finding proofs. Solved Exercise 1.1 is a good example of the process of finding a proof. You start by ensuring you understand what the statement means, and then come up with an informal argument why it should be true. You then transform the informal argument into a rigorous proof. This proof need not be very long or overly formal, but should clearly establish why the conclusion of the statement follows from its assumptions.

The concepts of degrees and connectivity extend naturally to *directed graphs*, defined as follows.

Definition 1.8 — Directed graphs. A *directed graph* $G = (V, E)$ consists of a set V and a set $E \subseteq V \times V$ of *ordered pairs* of V. We sometimes denote the edge (u, v) also as $u \to v$. If the edge $u \to v$ is present in the graph then we say that v is an *out-neighbor* of u and u is an *in-neighbor* of v.

A directed graph might contain both $u \to v$ and $v \to u$ in which case u will be both an in-neighbor and an out-neighbor of v and vice versa. The *in-degree* of u is the number of in-neighbors it has, and the

out-degree of v is the number of out-neighbors it has. A *path* in the graph is a tuple $(u_0, \ldots, u_k) \in V^{k+1}$, for some $k > 0$ such that u_{i+1} is an out-neighbor of u_i for every $i \in [k]$. As in the undirected case, a *simple path* is a path $(u_0, \ldots, u_{k-1})$ where all the u_i's are distinct and a *cycle* is a path $(u_0, \ldots, u_k)$ where $u_0 = u_k$. One type of directed graphs we often care about is *directed acyclic graphs* or *DAGs*, which, as their name implies, are directed graphs without any cycles:

Definition 1.9 — Directed Acyclic Graphs. We say that $G = (V, E)$ is a *directed acyclic graph* (*DAG*) if it is a directed graph and there does not exist a list of vertices $u_0, u_1, \ldots, u_k \in V$ such that $u_0 = u_k$ and for every $i \in [k]$, the edge $u_i \to u_{i+1}$ is in E.

The lemmas we mentioned above have analogs for directed graphs. We again leave the proofs (which are essentially identical to their undirected analogs) as exercises.

Lemma 1.10 In any directed graph $G = (V, E)$, the sum of the in-degrees is equal to the sum of the out-degrees, which is equal to the number of edges.

Lemma 1.11 In any directed graph G, if there is a path from u to v and a path from v to w, then there is a path from u to w.

Lemma 1.12 For every directed graph $G = (V, E)$ and a pair u, v such that there is a path from u to v, the *shortest path* from u to v is simple.

Remark 1.13 — Labeled graphs. For some applications we will consider *labeled graphs*, where the vertices or edges have associated *labels* (which can be numbers, strings, or members of some other set). We can think of such a graph as having an associated (possibly partial) *labelling function* $L : V \cup E \to \mathcal{L}$, where $\mathcal{L}$ is the set of potential labels. However we will typically not refer explicitly to this labeling function and simply say things such as "vertex v has the label α".

1.4.5 Logic operators and quantifiers

If P and Q are some statements that can be true or false, then P AND Q (denoted as $P \wedge Q$) is a statement that is true if and only if both P *and* Q are true, and P OR Q (denoted as $P \vee Q$) is a statement that is true if and only if either P *or* Q is true. The *negation* of P, denoted as $\neg P$ or $\overline{P}$, is true if and only if P is false.

Suppose that $P(x)$ is a statement that depends on some *parameter* x (also sometimes known as an *unbound* variable) in the sense that for every instantiation of x with a value from some set S, $P(x)$ is either

true or false. For example, $x > 7$ is a statement that is not a priori true or false, but becomes true or false whenever we instantiate x with some real number. We denote by $\forall_{x \in S} P(x)$ the statement that is true if and only if $P(x)$ is true *for every* $x \in S$.[4] We denote by $\exists_{x \in S} P(x)$ the statement that is true if and only if *there exists* some $x \in S$ such that $P(x)$ is true.

[4] In this book, we place the variable bound by a quantifier in a subscript and so write $\forall_{x \in S} P(x)$. Many other texts do not use this subscript notation and so will write the same statement as $\forall x \in S,\ P(x)$.

For example, the following is a formalization of the true statement that there exists a natural number n larger than 100 that is not divisible by 3:

$$\exists_{n \in \mathbb{N}} (n > 100) \wedge (\forall_{k \in N} k + k + k \neq n) \ .$$

"For sufficiently large n." One expression that we will see come up time and again in this book is the claim that some statement $P(n)$ is true "for sufficiently large n". What this means is that there exists an integer N_0 such that $P(n)$ is true for every $n > N_0$. We can formalize this as $\exists_{N_0 \in \mathbb{N}} \forall_{n > N_0} P(n)$.

1.4.6 Quantifiers for summations and products

The following shorthands for summing up or taking products of several numbers are often convenient. If $S = \{s_0, \ldots, s_{n-1}\}$ is a finite set and $f : S \to \mathbb{R}$ is a function, then we write $\sum_{x \in S} f(x)$ as shorthand for

$$f(s_0) + f(s_1) + f(s_2) + \ldots + f(s_{n-1}) \ ,$$

and $\prod_{x \in S} f(x)$ as shorthand for

$$f(s_0) \cdot f(s_1) \cdot f(s_2) \cdot \ldots \cdot f(s_{n-1}) \ .$$

For example, the sum of the squares of all numbers from 1 to 100 can be written as

$$\sum_{i \in \{1,\ldots,100\}} i^2 \ . \tag{1.1}$$

Since summing up over intervals of integers is so common, there is a special notation for it. For every two integers, $a \leq b$, $\sum_{i=a}^{b} f(i)$ denotes $\sum_{i \in S} f(i)$ where $S = \{x \in \mathbb{Z} : a \leq x \leq b\}$. Hence, we can write the sum (1.1) as

$$\sum_{i=1}^{100} i^2 \ .$$

1.4.7 Parsing formulas: bound and free variables

In mathematics, as in coding, we often have symbolic "variables" or "parameters". It is important to be able to understand, given some formula, whether a given variable is *bound* or *free* in this formula. For

example, in the following statement n is free but a and b are bound by the $\exists$ quantifier:

$$\exists_{a,b\in\mathbb{N}}(a \neq 1) \wedge (a \neq n) \wedge (n = a \times b) \tag{1.2}$$

Since n is free, it can be set to any value, and the truth of the statement (1.2) depends on the value of n. For example, if $n = 8$ then (1.2) is true, but for $n = 11$ it is false. (Can you see why?)

The same issue appears when parsing code. For example, in the following snippet from the C programming language

```
for (int i=0 ; i<n ; i=i+1) {
    printf("*");
}
```

the variable `i` is bound within the `for` block but the variable `n` is free.

The main property of bound variables is that we can *rename* them (as long as the new name doesn't conflict with another used variable) without changing the meaning of the statement. Thus for example the statement

$$\exists_{x,y\in\mathbb{N}}(x \neq 1) \wedge (x \neq n) \wedge (n = x \times y) \tag{1.3}$$

is *equivalent* to (1.2) in the sense that it is true for exactly the same set of n's.

Similarly, the code

```
for (int j=0 ; j<n ; j=j+1) {
    printf("*");
}
```

produces the same result as the code above that used `i` instead of `j`.

Remark 1.14 — Aside: mathematical vs programming notation. Mathematical notation has a lot of similarities with programming language, and for the same reasons. Both are formalisms meant to convey complex concepts in a precise way. However, there are some cultural differences. In programming languages, we often try to use meaningful variable names such as `NumberOfVertices` while in math we often use short identifiers such as n. Part of it might have to do with the tradition of mathematical proofs as being handwritten and verbally presented, as opposed to typed up and compiled. Another reason is if the wrong variable name is used in a proof, at worst it causes confusion to readers; when the wrong variable name

is used in a program, planes might crash, patients might die, and rockets could explode.

One consequence of that is that in mathematics we often end up reusing identifiers, and also "run out" of letters and hence use Greek letters too, as well as distinguish between small and capital letters and different font faces. Similarly, mathematical notation tends to use quite a lot of "overloading", using operators such as $+$ for a great variety of objects (e.g., real numbers, matrices, finite field elements, etc..), and assuming that the meaning can be inferred from the context.

Both fields have a notion of "types", and in math we often try to reserve certain letters for variables of a particular type. For example, variables such as i, j, k, ℓ, m, n will often denote integers, and ϵ will often denote a small positive real number (see Section 1.7 for more on these conventions). When reading or writing mathematical texts, we usually don't have the advantage of a "compiler" that will check type safety for us. Hence it is important to keep track of the type of each variable, and see that the operations that are performed on it "make sense".

Kun's book [Kun18] contains an extensive discussion on the similarities and differences between the cultures of mathematics and programming.

1.4.8 Asymptotics and Big-O notation

"$\log \log \log n$ has been proved to go to infinity, but has never been observed to do so.", Anonymous, quoted by Carl Pomerance (2000)

It is often very cumbersome to describe precisely quantities such as running time and is also not needed, since we are typically mostly interested in the "higher order terms". That is, we want to understand the *scaling behavior* of the quantity as the input variable grows. For example, as far as running time goes, the difference between an n^5-time algorithm and an n^2-time one is much more significant than the difference between a $100n^2 + 10n$ time algorithm and a $10n^2$ time algorithm. For this purpose, O-notation is extremely useful as a way to "declutter" our text and focus our attention on what really matters. For example, using O-notation, we can say that both $100n^2 + 10n$ and $10n^2$ are simply $\Theta(n^2)$ (which informally means "the same up to constant factors"), while $n^2 = o(n^5)$ (which informally means that n^2 is "much smaller than" n^5).

Generally (though still informally), if F, G are two functions mapping natural numbers to non-negative reals, then "$F = O(G)$" means that $F(n) \leq G(n)$ if we don't care about constant factors, while "$F = o(G)$" means that F is much smaller than G, in the sense that no matter by what constant factor we multiply F, if we take n to be large

enough then G will be bigger (for this reason, sometimes $F = o(G)$ is written as $F \ll G$). We will write $F = \Theta(G)$ if $F = O(G)$ and $G = O(F)$, which one can think of as saying that F is the same as G if we don't care about constant factors. More formally, we define Big-O notation as follows:

Definition 1.15 — Big-O notation. Let $\mathbb{R}_+ = \{x \in \mathbb{R} \mid x > 0\}$ be the set of positive real numbers. For two functions $F, G : \mathbb{N} \to \mathbb{R}_+$, we say that $F = O(G)$ if there exist numbers $a, N_0 \in \mathbb{N}$ such that $F(n) \leq a \cdot G(n)$ for every $n > N_0$. We say that $F = \Theta(G)$ if $F = O(G)$ and $G = O(F)$. We say that $F = \Omega(G)$ if $G = O(F)$.

We say that $F = o(G)$ if for every $\epsilon > 0$ there is some N_0 such that $F(n) < \epsilon G(n)$ for every $n > N_0$. We say that $F = \omega(G)$ if $G = o(F)$.

It's often convenient to use "anonymous functions" in the context of O-notation. For example, when we write a statement such as $F(n) = O(n^3)$, we mean that $F = O(G)$ where G is the function defined by $G(n) = n^3$. Chapter 7 in Jim Apsnes' notes on discrete math provides a good summary of O notation; see also this tutorial for a gentler and more programmer-oriented introduction.

O is not equality. Using the equality sign for O-notation is extremely common, but is somewhat of a misnomer, since a statement such as $F = O(G)$ really means that F is in the set $\{G' : \exists_{N,c} \text{ s.t. } \forall_{n>N} G'(n) \leq cG(n)\}$. If anything, it makes more sense to use *inequalities* and write $F \leq O(G)$ and $F \geq \Omega(G)$, reserving equality for $F = \Theta(G)$, and so we will sometimes use this notation too, but since the equality notation is quite firmly entrenched we often stick to it as well. (Some texts write $F \in O(G)$ instead of $F = O(G)$, but we will not use this notation.) Despite the misleading equality sign, you should remember that a statement such as $F = O(G)$ means that F is "at most" G in some rough sense when we ignore constants, and a statement such as $F = \Omega(G)$ means that F is "at least" G in the same rough sense.

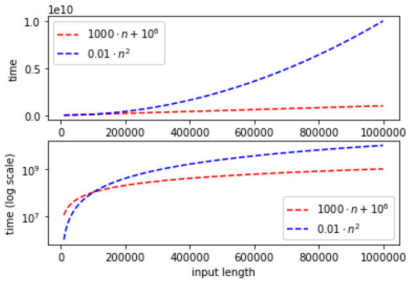


Figure 1.7: If $F(n) = o(G(n))$ then for sufficiently large n, $F(n)$ will be smaller than $G(n)$. For example, if Algorithm A runs in time $1000 \cdot n + 10^6$ and Algorithm B runs in time $0.01 \cdot n^2$ then even though B might be more efficient for smaller inputs, when the inputs get sufficiently large, A will run *much* faster than B.

1.4.9 Some "rules of thumb" for Big-O notation

There are some simple heuristics that can help when trying to compare two functions F and G:

- Multiplicative constants don't matter in O-notation, and so if $F(n) = O(G(n))$ then $100F(n) = O(G(n))$.
- When adding two functions, we only care about the larger one. For example, for the purpose of O-notation, $n^3 + 100n^2$ is the same as n^3, and in general in any polynomial, we only care about the larger exponent.

- For every two constants $a, b > 0$, $n^a = O(n^b)$ if and only if $a \leq b$, and $n^a = o(n^b)$ if and only if $a < b$. For example, combining the two observations above, $100n^2 + 10n + 100 = o(n^3)$.

- Polynomial is always smaller than exponential: $n^a = o(2^{n^\epsilon})$ for every two constants $a > 0$ and $\epsilon > 0$ even if ϵ is much smaller than a. For example, $100n^{100} = o(2^{\sqrt{n}})$.

- Similarly, logarithmic is always smaller than polynomial: $(\log n)^a$ (which we write as $\log^a n$) is $o(n^\epsilon)$ for every two constants $a, \epsilon > 0$. For example, combining the observations above, $100n^2 \log^{100} n = o(n^3)$.

Remark 1.16 — Big O for other applications (optional). While Big-O notation is often used to analyze running time of algorithms, this is by no means the only application. We can use O notation to bound asymptotic relations between any functions mapping integers to positive numbers. It can be used regardless of whether these functions are a measure of running time, memory usage, or any other quantity that may have nothing to do with computation. Here is one example which is unrelated to this book (and hence one that you can feel free to skip): one way to state the Riemann Hypothesis (one of the most famous open questions in mathematics) is that it corresponds to the conjecture that the number of primes between 0 and n is equal to $\int_2^n \frac{1}{\ln x} dx$ up to an additive error of magnitude at most $O(\sqrt{n} \log n)$.

1.5 PROOFS

Many people think of mathematical proofs as a sequence of logical deductions that starts from some axioms and ultimately arrives at a conclusion. In fact, some dictionaries define proofs that way. This is not entirely wrong, but at its essence, a mathematical proof of a statement X is simply an argument that convinces the reader that X is true beyond a shadow of a doubt.

To produce such a proof you need to:

1. Understand precisely what X means.

2. Convince *yourself* that X is true.

3. Write your reasoning down in plain, precise and concise English (using formulas or notation only when they help clarity).

In many cases, the first part is the most important one. Understanding what a statement means is oftentimes more than halfway towards understanding why it is true. In the third part, to convince the reader beyond a shadow of a doubt, we will often want to break down the reasoning to "basic steps", where each basic step is simple enough to be "self-evident". The combination of all steps yields the desired statement.

1.5.1 Proofs and programs

There is a great deal of similarity between the process of writing *proofs* and that of writing *programs*, and both require a similar set of skills. Writing a *program* involves:

1. Understanding what is the *task* we want the program to achieve.
2. Convincing *yourself* that the task can be achieved by a computer, perhaps by planning on a whiteboard or notepad how you will break it up into simpler tasks.
3. Converting this plan into code that a compiler or interpreter can understand, by breaking up each task into a sequence of the basic operations of some programming language.

In programs as in proofs, step 1 is often the most important one. A key difference is that the reader for proofs is a human being and the reader for programs is a computer. (This difference is eroding with time as more proofs are being written in a *machine verifiable* form; moreover, to ensure correctness and maintainability of programs, it is important that they can be read and understood by humans.) Thus our emphasis is on *readability* and having a *clear logical flow* for our proof (which is not a bad idea for programs as well). When writing a proof, you should think of your audience as an intelligent but highly skeptical and somewhat petty reader, that will "call foul" at every step that is not well justified.

1.5.2 Proof writing style

A mathematical proof is a piece of writing, but it is a specific genre of writing with certain conventions and preferred styles. As in any writing, practice makes perfect, and it is also important to revise your drafts for clarity.

In a proof for the statement X, all the text between the words **"Proof:"** and **"QED"** should be focused on establishing that X is true. Digressions, examples, or ruminations should be kept outside these two words, so they do not confuse the reader. The proof should have a clear logical flow in the sense that every sentence or equation in it should have some purpose and it should be crystal-clear to the reader

what this purpose is. When you write a proof, for every equation or sentence you include, ask yourself:

1. Is this sentence or equation stating that some statement is true?

2. If so, does this statement follow from the previous steps, or are we going to establish it in the next step?

3. What is the *role* of this sentence or equation? Is it one step towards proving the original statement, or is it a step towards proving some intermediate claim that you have stated before?

4. Finally, would the answers to questions 1-3 be clear to the reader? If not, then you should reorder, rephrase, or add explanations.

Some helpful resources on mathematical writing include this handout by Lee, this handout by Hutching, as well as several of the excellent handouts in Stanford's CS 103 class.

1.5.3 Patterns in proofs

"If it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic.", Lewis Carroll, *Through the looking-glass*.

Just like in programming, there are several common patterns of proofs that occur time and again. Here are some examples:

Proofs by contradiction: One way to prove that X is true is to show that if X was false it would result in a contradiction. Such proofs often start with a sentence such as "Suppose, towards a contradiction, that X is false" and end with deriving some contradiction (such as a violation of one of the assumptions in the theorem statement). Here is an example:

Lemma 1.17 There are no natural numbers a, b such that $\sqrt{2} = \frac{a}{b}$.

Proof. Suppose, towards a contradiction that this is false, and so let $a \in \mathbb{N}$ be the smallest number such that there exists some $b \in \mathbb{N}$ satisfying $\sqrt{2} = \frac{a}{b}$. Squaring this equation we get that $2 = a^2/b^2$ or $a^2 = 2b^2$ $(*)$. But this means that a^2 is *even*, and since the product of two odd numbers is odd, it means that a is even as well, or in other words, $a = 2a'$ for some $a' \in \mathbb{N}$. Yet plugging this into $(*)$ shows that $4a'^2 = 2b^2$ which means $b^2 = 2a'^2$ is an even number as well. By the same considerations as above we get that b is even and hence $a/2$ and $b/2$ are two natural numbers satisfying $\frac{a/2}{b/2} = \sqrt{2}$, contradicting the minimality of a.

■

Proofs of a universal statement: Often we want to prove a statement X of the form "Every object of type O has property P." Such proofs often start with a sentence such as "Let o be an object of type O" and end by showing that o has the property P. Here is a simple example:

Lemma 1.18 For every natural number $n \in N$, either n or $n+1$ is even.

Proof. Let $n \in N$ be some number. If $n/2$ is a whole number then we are done, since then $n = 2(n/2)$ and hence it is even. Otherwise, $n/2 + 1/2$ is a whole number, and hence $2(n/2 + 1/2) = n + 1$ is even. ■

Proofs of an implication: Another common case is that the statement X has the form "A implies B". Such proofs often start with a sentence such as "Assume that A is true" and end with a derivation of B from A. Here is a simple example:

Lemma 1.19 If $b^2 \geq 4ac$ then there is a solution to the quadratic equation $ax^2 + bx + c = 0$.

Proof. Suppose that $b^2 \geq 4ac$. Then $d = b^2 - 4ac$ is a non-negative number and hence it has a square root s. Thus $x = (-b + s)/(2a)$ satisfies

$$\begin{aligned} ax^2 + bx + c &= a(-b+s)^2/(4a^2) + b(-b+s)/(2a) + c \\ &= (b^2 - 2bs + s^2)/(4a) + (-b^2 + bs)/(2a) + c \,. \end{aligned} \tag{1.4}$$

■

Rearranging the terms of (1.4) we get

$$s^2/(4a) + c - b^2/(4a) = (b^2 - 4ac)/(4a) + c - b^2/(4a) = 0$$

Proofs of equivalence: If a statement has the form "A if and only if B" (often shortened as "A iff B") then we need to prove both that A implies B and that B implies A. We call the implication that A implies B the "only if" direction, and the implication that B implies A the "if" direction.

Proofs by combining intermediate claims: When a proof is more complex, it is often helpful to break it apart into several steps. That is, to prove the statement X, we might first prove statements X_1,X_2, and X_3 and then prove that $X_1 \wedge X_2 \wedge X_3$ implies X. (Recall that $\wedge$ denotes the logical AND operator.)

Proofs by case distinction: This is a special case of the above, where to prove a statement X we split into several cases $C_1, \ldots, C_k$, and prove that **(a)** the cases are *exhaustive*, in the sense that *one* of the cases C_i must happen and **(b)** go one by one and prove that each one of the cases C_i implies the result X that we are after.

Proofs by induction: We discuss induction and give an example in Section 1.6.1 below. We can think of such proofs as a variant of the above, where we have an unbounded number of intermediate claims $X_0, X_1, X_2, \ldots, X_k$, and we prove that X_0 is true, as well as that X_0 implies X_1, and that $X_0 \wedge X_1$ implies X_2, and so on and so forth. The website for CMU course 15-251 contains a useful handout on potential pitfalls when making proofs by induction.

"Without loss of generality (w.l.o.g)": This term can be initially quite confusing. It is essentially a way to simplify proofs by case distinctions. The idea is that if Case 1 is equal to Case 2 up to a change of variables or a similar transformation, then the proof of Case 1 will also imply the proof of Case 2. It is always a statement that should be viewed with suspicion. Whenever you see it in a proof, ask yourself if you understand *why* the assumption made is truly without loss of generality, and when you use it, try to see if the use is indeed justified. When writing a proof, sometimes it might be easiest to simply repeat the proof of the second case (adding a remark that the proof is very similar to the first one).

R

Remark 1.20 — Hierarchical Proofs (optional). Mathematical proofs are ultimately written in English prose. The well-known computer scientist Leslie Lamport argues that this is a problem, and proofs should be written in a more formal and rigorous way. In his manuscript he proposes an approach for *structured hierarchical proofs*, that have the following form:

- A proof for a statement of the form "If A then B" is a sequence of numbered claims, starting with the assumption that A is true, and ending with the claim that B is true.
- Every claim is followed by a proof showing how it is derived from the previous assumptions or claims.
- The proof for each claim is itself a sequence of subclaims.

The advantage of Lamport's format is that the role that every sentence in the proof plays is very clear. It is also much easier to transform such proofs into machine-checkable forms. The disadvantage is that such proofs can be tedious to read and write, with less differentiation between the important parts of the arguments versus the more routine ones.

1.6 EXTENDED EXAMPLE: TOPOLOGICAL SORTING

In this section we will prove the following: every directed acyclic graph (DAG, see Definition 1.9) can be arranged in layers so that for all directed edges $u \to v$, the layer of v is larger than the layer of u. This result is known as topological sorting and is used in many applications, including task scheduling, build systems, software package management, spreadsheet cell calculations, and many others (see Fig. 1.8). In fact, we will also use it ourselves later on in this book.

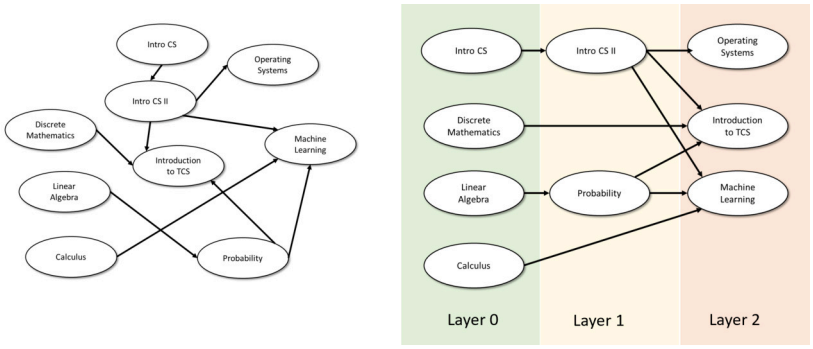


Figure 1.8: An example of *topological sorting*. We consider a directed graph corresponding to a prerequisite graph of the courses in some Computer Science program. The edge $u \to v$ means that the course u is a prerequisite for the course v. A *layering* or "topological sorting" of this graph is the same as mapping the courses to semesters so that if we decide to take the course v in semester $f(v)$, then we have already taken all the prerequisites for v (i.e., its in-neighbors) in prior semesters.

We start with the following definition. A *layering* of a directed graph is a way to assign for every vertex v a natural number (corresponding to its layer), such that v's in-neighbors are in lower-numbered layers than v, and v's out-neighbors are in higher-numbered layers. The formal definition is as follows:

Definition 1.21 — Layering of a DAG. Let $G = (V, E)$ be a directed graph. A *layering* of G is a function $f : V \to \mathbb{N}$ such that for every edge $u \to v$ of G, $f(u) < f(v)$.

In this section we prove that a directed graph is acyclic if and only if it has a valid layering.

Theorem 1.22 — Topological Sort. Let G be a directed graph. Then G is acyclic if and only if there exists a layering f of G.

To prove such a theorem, we need to first understand what it means. Since it is an "if and only if" statement, Theorem 1.22 corresponds to two statements:

Lemma 1.23 For every directed graph G, if G is acyclic then it has a layering.

Lemma 1.24 For every directed graph G, if G has a layering, then it is acyclic.

To prove Theorem 1.22 we need to prove both Lemma 1.23 and Lemma 1.24. Lemma 1.24 is actually not that hard to prove. Intuitively, if G contains a *cycle*, then it cannot be the case that all edges on the cycle increase in layer number, since if we travel along the cycle at some point we must come back to the place we started from. The formal proof is as follows:

Proof. Let $G = (V, E)$ be a directed graph and let $f : V \to \mathbb{N}$ be a layering of G as per Definition 1.21 . Suppose, towards a contradiction, that G is not acyclic, and hence there exists some cycle $u_0, u_1, \ldots, u_k$ such that $u_0 = u_k$ and for every $i \in [k]$ the edge $u_i \to u_{i+1}$ is present in G. Since f is a layering, for every $i \in [k]$, $f(u_i) < f(u_{i+1})$, which means that

$$f(u_0) < f(u_1) < \cdots < f(u_k)$$

but this is a contradiction since $u_0 = u_k$ and hence $f(u_0) = f(u_k)$. ■

Lemma 1.23 corresponds to the more difficult (and useful) direction. To prove it, we need to show how, given an arbitrary DAG G, we can come up with a layering of the vertices of G so that all edges "go up".

P

If you have not seen the proof of this theorem before (or don't remember it), this would be an excellent point to pause and try to prove it yourself. One way to do it would be to describe an *algorithm* that given as input a directed acyclic graph G on n vertices and $n-2$ or fewer edges, constructs an array F of length n such that for every edge $u \to v$ in the graph $F[u] < F[v]$.

1.6.1 Mathematical induction

There are several ways to prove Lemma 1.23. One approach to do is to start by proving it for small graphs, such as graphs with 1, 2 or 3 vertices (see Fig. 1.9, for which we can check all the cases, and then try to extend the proof for larger graphs). The technical term for this proof approach is *proof by induction*.

Induction is simply an application of the self-evident Modus Ponens rule that says that if

(a) P is true
and

(b) P implies Q
then Q is true.

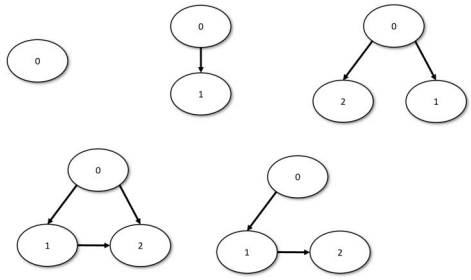


Figure 1.9: Some examples of DAGs of one, two and three vertices, and valid ways to assign layers to the vertices.

In the setting of proofs by induction we typically have a statement $Q(k)$ that is parameterized by some integer k, and we prove that **(a)** $Q(0)$ is true, and **(b)** For every $k > 0$, if $Q(0), \ldots, Q(k-1)$ are all true then $Q(k)$ is true. (Usually proving **(b)** is the hard part, though there are examples where the "base case" **(a)** is quite subtle.) By applying Modus Ponens, we can deduce from **(a)** and **(b)** that $Q(1)$ is true. Once we did so, since we now know that both $Q(0)$ and $Q(1)$ are true, then we can use this and **(b)** to deduce (again using Modus Ponens) that $Q(2)$ is true. We can repeat the same reasoning again and again to obtain that $Q(k)$ is true for every k. The statement **(a)** is called the "base case", while **(b)** is called the "inductive step". The assumption in **(b)** that $Q(i)$ holds for $i < k$ is called the "inductive hypothesis". (The form of induction described here is sometimes called "strong induction" as opposed to "weak induction" where we replace **(b)** by the statement **(b')** that if $Q(k-1)$ is true then $Q(k)$ is true; weak induction can be thought of as the special case of strong induction where we don't use the assumption that $Q(0), \ldots, Q(k-2)$ are true.)

Remark 1.25 — Induction and recursion. Proofs by induction are closely related to algorithms by recursion. In both cases we reduce solving a larger problem to solving a smaller instance of itself. In a recursive algorithm to solve some problem P on an input of length k we ask ourselves "what if someone handed me a way to solve P on instances smaller than k?". In an inductive proof to prove a statement Q parameterized by a number k, we ask ourselves "what if I already knew that $Q(k')$ is true for $k' < k$?". Both induction and recursion are crucial concepts for this course and Computer Science at large (and even other areas of inquiry, including not just mathematics but other sciences as well). Both can be confusing at first, but with time and practice they become clearer. For more on proofs by induction and recursion, you might find the following Stanford CS 103 handout, this MIT 6.00 lecture or this excerpt of the Lehman-Leighton book useful.

1.6.2 Proving the result by induction

There are several ways to prove Lemma 1.23 by induction. We will use induction on the number n of vertices, and so we will define the statement $Q(n)$ as follows:

$Q(n)$ is *"For every DAG $G = (V, E)$ with n vertices, there is a layering of G."*

The statement for $Q(0)$ (where the graph contains no vertices) is trivial. Thus it will suffice to prove the following: *for every $n > 0$, if $Q(n-1)$ is true then $Q(n)$ is true.*

To do so, we need to somehow find a way, given a graph G of n vertices, to reduce the task of finding a layering for G into the task of finding a layering for some other graph G' of $n-1$ vertices. The idea is that we will find a *source* of G: a vertex v that has no in-neighbors. We can then assign to v the layer 0, and layer the remaining vertices using the inductive hypothesis in layers $1, 2, \ldots$.

The above is the intuition behind the proof of Lemma 1.23, but when writing the formal proof below, we use the benefit of hindsight, and try to streamline what was a messy journey into a linear and easy-to-follow flow of logic that starts with the word **"Proof:"** and ends with **"QED"** or the symbol ■.[5] Discussions, examples and digressions can be very insightful, but we keep them outside the space delimited between these two words, where (as described by this excellent handout) "every sentence must be load-bearing". Just like we do in programming, we can break the proof into little "subroutines" or "functions" (known as *lemmas* or *claims* in math language), which will be smaller statements that help us prove the main result. However, the proof should be structured in a way that ensures that it is always crystal-clear to the reader in what stage we are of the proof. The reader should be able to tell what the role of every sentence is in the proof and which part it belongs to. We now present the formal proof of Lemma 1.23.

[5] QED stands for "quod erat demonstrandum", which is Latin for "what was to be demonstrated" or "the very thing it was required to have shown".

Proof of Lemma 1.23. Let $G = (V, E)$ be a DAG and $n = |V|$ be the number of its vertices. We prove the lemma by induction on n. The base case is $n = 0$ where there are no vertices, and so the statement is trivially true.[6] For the case of $n > 0$, we make the inductive hypothesis that every DAG G' of at most $n-1$ vertices has a layering.

We make the following claim:

Claim: G must contain a vertex v of in-degree zero.

Proof of Claim: Suppose otherwise that every vertex $v \in V$ has an in-neighbor. Let v_0 be some vertex of G, let v_1 be an in-neighbor of v_0, v_2 be an in-neighbor of v_1, and continue in this way for n steps until we construct a list $v_0, v_1, \ldots, v_n$ such that for every $i \in [n]$, v_{i+1} is an in-neighbor of v_i, or in other words the edge $v_{i+1} \to v_i$ is present in the graph. Since there are only n vertices in this graph, one of the $n+1$ vertices in this sequence must repeat itself, and so there exists $i < j$ such that $v_i = v_j$. But then the sequence $v_j \to v_{j-1} \to \cdots \to v_i$ is a cycle in G, contradicting our assumption that it is acyclic. (**QED Claim**)

Given the claim, we can let v_0 be some vertex of in-degree zero in G, and let G' be the graph obtained by removing v_0 from G. G' has

[6] Using $n = 0$ as the base case is logically valid, but can be confusing. If you find the trivial $n = 0$ case to be confusing, you can always directly verify the statement for $n = 1$ and then use both $n = 0$ and $n = 1$ as the base cases.

$n-1$ vertices and hence per the inductive hypothesis has a layering $f' : (V \setminus \{v_0\}) \to \mathbb{N}$. We define $f : V \to \mathbb{N}$ as follows:

$$f(v) = \begin{cases} f'(v)+1 & v \neq v_0 \\ 0 & v = v_0 \end{cases} .$$

We claim that f is a valid layering, namely that for every edge $u \to v$, $f(u) < f(v)$. To prove this, we split into cases:

- **Case 1:** $u \neq v_0, v \neq v_0$. In this case the edge $u \to v$ exists in the graph G' and hence by the inductive hypothesis $f'(u) < f'(v)$ which implies that $f'(u)+1 < f'(v)+1$.
- **Case 2:** $u = v_0, v \neq v_0$. In this case $f(u) = 0$ and $f(v) = f'(v)+1 > 0$.
- **Case 3:** $u \neq v_0, v = v_0$. This case can't happen since v_0 does not have in-neighbors.
- **Case 4:** $u = v_0, v = v_0$. This case again can't happen since it means that v_0 is its own-neighbor — it is involved in a *self loop* which is a form cycle that is disallowed in an acyclic graph.

Thus, f is a valid layering for G which completes the proof. ■

Reading a proof is no less of an important skill than producing one. In fact, just like understanding code, it is a highly non-trivial skill in itself. Therefore I strongly suggest that you re-read the above proof, asking yourself at every sentence whether the assumption it makes is justified, and whether this sentence truly demonstrates what it purports to achieve. Another good habit is to ask yourself when reading a proof for every variable you encounter (such as u, i, G', f', etc. in the above proof) the following questions: **(1)** What *type* of variable is it? Is it a number? a graph? a vertex? a function? and **(2)** What do we know about it? Is it an arbitrary member of the set? Have we shown some facts about it?, and **(3)** What are we *trying* to show about it?.

1.6.3 Minimality and uniqueness

Theorem 1.22 guarantees that for every DAG $G = (V, E)$ there exists some layering $f : V \to \mathbb{N}$ but this layering is not necessarily *unique*. For example, if $f : V \to \mathbb{N}$ is a valid layering of the graph then so is the function f' defined as $f'(v) = 2 \cdot f(v)$. However, it turns out that

the *minimal* layering is unique. A minimal layering is one where every vertex is given the smallest layer number possible. We now formally define minimality and state the uniqueness theorem:

Theorem 1.26 — Minimal layering is unique. Let $G = (V, E)$ be a DAG. We say that a layering $f : V \to \mathbb{N}$ is *minimal* if for every vertex $v \in V$, if v has no in-neighbors then $f(v) = 0$ and if v has in-neighbors then there exists an in-neighbor u of v such that $f(u) = f(v) - 1$.

For every layering $f, g : V \to \mathbb{N}$ of G, if both f and g are minimal then $f = g$.

The definition of minimality in Theorem 1.26 implies that for every vertex $v \in V$, we cannot move it to a lower layer without making the layering invalid. If v is a source (i.e., has in-degree zero) then a minimal layering f must put it in layer 0, and for every other v, if $f(v) = i$, then we cannot modify this to set $f(v) \leq i - 1$ since there is an-neighbor u of v satisfying $f(u) = i - 1$. What Theorem 1.26 says is that a minimal layering f is *unique* in the sense that every other minimal layering is equal to f.

Proof Idea:

The idea is to prove the theorem by induction on the layers. If f and g are minimal then they must agree on the source vertices, since both f and g should assign these vertices to layer 0. We can then show that if f and g agree up to layer $i - 1$, then the minimality property implies that they need to agree in layer i as well. In the actual proof we use a small trick to save on writing. Rather than proving the statement that $f = g$ (or in other words that $f(v) = g(v)$ for every $v \in V$), we prove the weaker statement that $f(v) \leq g(v)$ for every $v \in V$. (This is a weaker statement since the condition that $f(v)$ is lesser or equal than to $g(v)$ is implied by the condition that $f(v)$ is equal to $g(v)$.) However, since f and g are just labels we give to two minimal layerings, by simply changing the names "f" and "g" the same proof also shows that $g(v) \leq f(v)$ for every $v \in V$ and hence that $f = g$.

⋆

Proof of Theorem 1.26. Let $G = (V, E)$ be a DAG and $f, g : V \to \mathbb{N}$ be two minimal valid layerings of G. We will prove that for every $v \in V$, $f(v) \leq g(v)$. Since we didn't assume anything about f, g except their minimality, the same proof will imply that for every $v \in V$, $g(v) \leq f(v)$ and hence that $f(v) = g(v)$ for every $v \in V$, which is what we needed to show.

We will prove that $f(v) \leq g(v)$ for every $v \in V$ by induction on $i = f(v)$. The case $i = 0$ is immediate: since in this case $f(v) = 0$, $g(v)$ must be at least $f(v)$. For the case $i > 0$, by the minimality of f,

if $f(v) = i$ then there must exist some in-neighbor u of v such that $f(u) = i - 1$. By the induction hypothesis we get that $g(u) \geq i - 1$, and since g is a valid layering it must hold that $g(v) > g(u)$ which means that $g(v) \geq i = f(v)$.

■

The proof of Theorem 1.26 is fully rigorous, but is written in a somewhat terse manner. Make sure that you read through it and understand *why* this is indeed an airtight proof of the Theorem's statement.

1.7 THIS BOOK: NOTATION AND CONVENTIONS

Most of the notation we use in this book is standard and is used in most mathematical texts. The main points where we diverge are:

- We index the natural numbers $\mathbb{N}$ starting with 0 (though many other texts, especially in computer science, do the same).
- We also index the set $[n]$ starting with 0, and hence define it as $\{0, \ldots, n-1\}$. In other texts it is often defined as $\{1, \ldots, n\}$. Similarly, we index our strings starting with 0, and hence a string $x \in \{0,1\}^n$ is written as $x_0 x_1 \cdots x_{n-1}$.
- If n is a natural number then 1^n does *not* equal the number 1 but rather this is the length n string $11 \cdots 1$ (that is a string of n ones). Similarly, 0^n refers to the length n string $00 \cdots 0$.
- *Partial* functions are functions that are not necessarily defined on all inputs. When we write $f : A \to B$ this means that f is a *total* function unless we say otherwise. When we want to emphasize that f can be a partial function, we will sometimes write $f : A \to_p B$.
- As we will see later on in the course, we will mostly describe our computational problems in terms of computing a *Boolean function* $f : \{0,1\}^* \to \{0,1\}$. In contrast, many other textbooks refer to the same task as *deciding a language* $L \subseteq \{0,1\}^*$. These two viewpoints are equivalent, since for every set $L \subseteq \{0,1\}^*$ there is a corresponding function F such that $F(x) = 1$ if and only if $x \in L$. Computing *partial functions* corresponds to the task known in the literature as a solving *promise problem*. Because the language notation is so prevalent in other textbooks, we will occasionally remind the reader of this correspondence.
- We use $\lceil x \rceil$ and $\lfloor x \rfloor$ for the "ceiling" and "floor" operators that correspond to "rounding up" or "rounding down" a number to the

nearest integer. We use $(x \mod y)$ to denote the "remainder" of x when divided by y. That is, $(x \mod y) = x - y\lfloor x/y \rfloor$. In context when an integer is expected we'll typically "silently round" the quantities to an integer. For example, if we say that x is a string of length $\sqrt{n}$ then this means that x is of length $\lceil \sqrt{n} \rceil$. (We round up for the sake of convention, but in most such cases, it will not make a difference whether we round up or down.)

- Like most Computer Science texts, we default to the logarithm in base two. Thus, $\log n$ is the same as $\log_2 n$.

- We will also use the notation $f(n) = poly(n)$ as a shorthand for $f(n) = n^{O(1)}$ (i.e., as shorthand for saying that there are some constants a, b such that $f(n) \leq a \cdot n^b$ for every sufficiently large n). Similarly, we will use $f(n) = polylog(n)$ as shorthand for $f(n) = poly(\log n)$ (i.e., as shorthand for saying that there are some constants a, b such that $f(n) \leq a \cdot (\log n)^b$ for every sufficiently large n).

- As is often the case in mathematical literature, we use the apostrophe character to enrich our set of identifiers. Typically if x denotes some object, then x', x'', etc. will denote other objects of the same type.

- To save on "cognitive load" we will often use round constants such as $10, 100, 1000$ in the statements of both theorems and problem set questions. When you see such a "round" constant, you can typically assume that it has no special significance and was just chosen arbitrarily. For example, if you see a theorem of the form "Algorithm A takes at most $1000 \cdot n^2$ steps to compute function F on inputs of length n" then probably the number 1000 is an arbitrary sufficiently large constant, and one could prove the same theorem with a bound of the form $c \cdot n^2$ for a constant c that is smaller than 1000. Similarly, if a problem asks you to prove that some quantity is at least $n/100$, it is quite possible that in truth the quantity is at least n/d for some constant d that is smaller than 100.

1.7.1 Variable name conventions

Like programming, mathematics is full of *variables*. Whenever you see a variable, it is always important to keep track of what its *type* is (e.g., whether the variable is a number, a string, a function, a graph, etc.). To make this easier, we try to stick to certain conventions and consistently use certain identifiers for variables of the same type. Some of these conventions are listed in Section 1.7.1 below. These conventions are not immutable laws and we might occasionally deviate from them.

Also, such conventions do not replace the need to explicitly declare for each new variable the type of object that it denotes.

Table 1.2: Conventions for identifiers in this book

Identifier	*Often denotes object of type*
i,j,k,ℓ,m,n	Natural numbers (i.e., in $\mathbb{N} = \{0, 1, 2, ...\}$)
ϵ, δ	Small positive real numbers (very close to 0)
x, y, z, w	Typically strings in $\{0,1\}^*$ though sometimes numbers or other objects. We often identify an object with its representation as a string.
G	A *graph*. The set of G's vertices is typically denoted by V. Often $V = [n]$. The set of G's edges is typically denoted by E.
S	Set
f, g, h	Functions. We often (though not always) use lowercase identifiers for *finite functions*, which map $\{0,1\}^n$ to $\{0,1\}^m$ (often $m = 1$).
F, G, H	Infinite (unbounded input) functions mapping $\{0,1\}^*$ to $\{0,1\}^*$ or $\{0,1\}^*$ to $\{0,1\}^m$ for some m. Based on context, the identifiers G, H are sometimes used to denote functions and sometimes graphs.
A, B, C	Boolean circuits
M, N	Turing machines
P, Q	Programs
T	A function mapping $\mathbb{N}$ to $\mathbb{N}$ that corresponds to a time bound.
c	A positive number (often an unspecified constant; e.g., $T(n) = O(n)$ corresponds to the existence of c s.t. $T(n) \leq c \cdot n$ every $n > 0$). We sometimes use a, b in a similar way.
Σ	Finite set (often used as the *alphabet* for a set of strings).

1.7.2 Some idioms

Mathematical texts often employ certain conventions or "idioms". Some examples of such idioms that we use in this text include the following:

- **"Let X be ..."**, **"let X denote ..."**, or **"let $X = ...$":** These are all different ways for us to say that we are *defining* the symbol X to stand for whatever expression is in the .... When X is a *property* of some objects we might define X by writing something along the lines of **"We say that** ... **has the property X if ...."**. While we often

try to define terms before they are used, sometimes a mathematical sentence reads easier if we use a term before defining it, in which case we add **"Where X is ..."** to explain how X is defined in the preceding expression.

- **Quantifiers:** Mathematical texts involve many quantifiers such as "for all" and "exists". We sometimes spell these in words as in **"for all** $i \in \mathbb{N}$**"** or **"there is** $x \in \{0,1\}^*$**"**, and sometimes use the formal symbols $\forall$ and $\exists$. It is important to keep track of which variable is quantified in what way the *dependencies* between the variables. For example, a sentence fragment such as **"for every** $k > 0$ **there exists** n**"** means that n can be chosen in a way that *depends* on k. The order of quantifiers is important. For example, the following is a true statement: *"for every natural number $k > 1$ there exists a prime number n such that n divides k."* In contrast, the following statement is false: *"there exists a prime number n such that for every natural number $k > 1$, n divides k."*

- **Numbered equations, theorems, definitions:** To keep track of all the terms we define and statements we prove, we often assign them a (typically numeric) label, and then refer back to them in other parts of the text.

- **(i.e.,), (e.g.,):** Mathematical texts tend to contain quite a few of these expressions. We use X (i.e., Y) in cases where Y is equivalent to X and X (e.g., Y) in cases where Y is an example of X (e.g., one can use phrases such as "a natural number (i.e., a non-negative integer)" or "a natural number (e.g., 7)").

- **"Thus"**, **"Therefore"** , **"We get that"**: This means that the following sentence is implied by the preceding one, as in "The n-vertex graph G is connected. Therefore it contains at least $n - 1$ edges." We sometimes use **"indeed"** to indicate that the following text justifies the claim that was made in the preceding sentence as in *"The n-vertex graph G has at least $n - 1$ edges. Indeed, this follows since G is connected."*

- **Constants:** In Computer Science, we typically care about how our algorithms' resource consumption (such as running time) *scales* with certain quantities (such as the length of the input). We refer to quantities that do not depend on the length of the input as *constants* and so often use statements such as *"there exists a constant $c > 0$ such that for every $n \in \mathbb{N}$, Algorithm A runs in at most $c \cdot n^2$ steps on inputs of length n."* The qualifier "constant" for c is not strictly needed but is added to emphasize that c here is a fixed number independent of n. In fact sometimes, to reduce cognitive load, we will simply replace c

by a sufficiently large round number such as 10, 100, or 1000, or use O-notation and write *"Algorithm A runs in $O(n^2)$ time."*

Chapter Recap

- The basic "mathematical data structures" we'll need are *numbers*, *sets*, *tuples*, *strings*, *graphs* and *functions*.
- We can use basic objects to define more complex notions. For example, *graphs* can be defined as a list of *pairs*.
- Given precise *definitions* of objects, we can state unambiguous and precise *statements*. We can then use mathematical *proofs* to determine whether these statements are true or false.
- A mathematical proof is not a formal ritual but rather a clear, precise and "bulletproof" argument certifying the truth of a certain statement.
- Big-O notation is an extremely useful formalism to suppress less significant details and allows us to focus on the high-level behavior of quantities of interest.
- The only way to get comfortable with mathematical notions is to apply them in the contexts of solving problems. You should expect to need to go back time and again to the definitions and notation in this chapter as you work through problems in this course.

1.8 EXERCISES

Exercise 1.1 — Logical expressions. a. Write a logical expression $\varphi(x)$ involving the variables x_0, x_1, x_2 and the operators $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT), such that $\varphi(x)$ is true if the majority of the inputs are *True*.

b. Write a logical expression $\varphi(x)$ involving the variables x_0, x_1, x_2 and the operators $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT), such that $\varphi(x)$ is true if the sum $\sum_{i=0}^{2} x_i$ (identifying "true" with 1 and "false" with 0) is *odd*.

■

Exercise 1.2 — Quantifiers. Use the logical quantifiers $\forall$ (for all), $\exists$ (there exists), as well as $\wedge, \vee, \neg$ and the arithmetic operations $+, \times, =, >, <$ to write the following:

a. An expression $\varphi(n, k)$ such that for every natural number n, k, $\varphi(n, k)$ is true if and only if k divides n.

b. An expression $\varphi(n)$ such that for every natural number n, $\varphi(n)$ is true if and only if n is a power of three.

Exercise 1.3 Describe the following statement in English words: $\forall_{n\in\mathbb{N}}\exists_{p>n}\forall a,b\in\mathbb{N}(a\times b\neq p)\vee(a=1)$.

Exercise 1.4 — Set construction notation. Describe in words the following sets:

a. $S=\{x\in\{0,1\}^{100}:\forall_{i\in\{0,\ldots,99\}}x_i=x_{99-i}\}$

b. $T=\{x\in\{0,1\}^*:\forall_{i,j\in\{2,\ldots,|x|-1\}}i\cdot j\neq|x|\}$

Exercise 1.5 — Existence of one to one mappings. For each one of the following pairs of sets (S,T), prove or disprove the following statement: there is a one to one function f mapping S to T.

a. Let $n>10$. $S=\{0,1\}^n$ and $T=[n]\times[n]\times[n]$.

b. Let $n>10$. S is the set of all functions mapping $\{0,1\}^n$ to $\{0,1\}$. $T=\{0,1\}^{n^3}$.

c. Let $n>100$. $S=\{k\in[n]\mid k \text{ is prime}\}$, $T=\{0,1\}^{\lceil\log n-1\rceil}$.

Exercise 1.6 — Inclusion Exclusion. a. Let A,B be finite sets. Prove that $|A\cup B|=|A|+|B|-|A\cap B|$.

b. Let $A_0,\ldots,A_{k-1}$ be finite sets. Prove that $|A_0\cup\cdots\cup A_{k-1}|\geq\sum_{i=0}^{k-1}|A_i|-\sum_{0\leq i<j<k}|A_i\cap A_j|$.

c. Let $A_0,\ldots,A_{k-1}$ be finite subsets of $\{1,\ldots,n\}$, such that $|A_i|=m$ for every $i\in[k]$. Prove that if $k>100n$, then there exist two distinct sets A_i,A_j s.t. $|A_i\cap A_j|\geq m^2/(10n)$.

Exercise 1.7 Prove that if S,T are finite and $F:S\to T$ is one to one then $|S|\leq|T|$.

Exercise 1.8 Prove that if S,T are finite and $F:S\to T$ is onto then $|S|\geq|T|$.

Exercise 1.9 Prove that for every finite S,T, there are $(|T|+1)^{|S|}$ partial functions from S to T.

Exercise 1.10 Suppose that $\{S_n\}_{n\in\mathbb{N}}$ is a sequence such that $S_0\leq 10$ and for $n>1$ $S_n\leq 5S_{\lfloor\frac{n}{5}\rfloor}+2n$. Prove by induction that $S_n\leq 100n\log n$ for every n.

Exercise 1.11 Prove that for every undirected graph G of 100 vertices, if every vertex has degree at most 4, then there exists a subset S of at least 20 vertices such that no two vertices in S are neighbors of one another.

Exercise 1.12 — O-notation. For every pair of functions F, G below, determine which of the following relations holds: $F = O(G)$, $F = \Omega(G)$, $F = o(G)$ or $F = \omega(G)$.

a. $F(n) = n$, $G(n) = 100n$.

b. $F(n) = n$, $G(n) = \sqrt{n}$.

c. $F(n) = n \log n$, $G(n) = 2^{(\log(n))^2}$.

d. $F(n) = \sqrt{n}$, $G(n) = 2^{\sqrt{\log n}}$.

e. $F(n) = \binom{n}{\lceil 0.2n \rceil}$, $G(n) = 2^{0.1n}$ (where $\binom{n}{k}$ is the number of k-sized subsets of a set of size n). See footnote for hint.[7]

[7] One way to do this is to use Stirling's approximation for the factorial function.

Exercise 1.13 Give an example of a pair of functions $F, G : \mathbb{N} \to \mathbb{N}$ such that neither $F = O(G)$ nor $G = O(F)$ holds.

Exercise 1.14 Prove that for every undirected graph G on n vertices, if G has at least n edges then G contains a cycle.

Exercise 1.15 Prove that for every undirected graph G of 1000 vertices, if every vertex has degree at most 4, then there exists a subset S of at least 200 vertices such that no two vertices in S are neighbors of one another.

1.9 BIBLIOGRAPHICAL NOTES

The heading "A Mathematician's Apology", refers to Hardy's classic book [Har41]. Even when Hardy is wrong, he is very much worth reading.

There are many online sources for the mathematical background needed for this book. In particular, the lecture notes for MIT 6.042 "Mathematics for Computer Science" [LLM18] are extremely comprehensive, and videos and assignments for this course are available online. Similarly, Berkeley CS 70: "Discrete Mathematics and Probability Theory" has extensive lecture notes online.

Other sources for discrete mathematics are Rosen [Ros19] and Jim Aspens' online book [Asp18]. Lewis and Zax [LZ19], as well as the online book of Fleck [Fle18], give a more gentle overview of much of the same material. Solow [Sol14] is a good introduction to proof reading and writing. Kun [Kun18] gives an introduction to mathematics aimed at readers with programming backgrounds. Stanford's CS 103 course has a wonderful collection of handouts on mathematical proof techniques and discrete mathematics.

The word *graph* in the sense of Definition 1.3 was coined by the mathematician Sylvester in 1878 in analogy with the chemical graphs used to visualize molecules. There is an unfortunate confusion between this term and the more common usage of the word "graph" as a way to plot data, and in particular a plot of some function $f(x)$ as a function of x. One way to relate these two notions is to identify every function $f : A \to B$ with the directed graph G_f over the vertex set $V = A \cup B$ such that G_f contains the edge $x \to f(x)$ for every $x \in A$. In a graph G_f constructed in this way, every vertex in A has out-degree equal to one. If the function f is *one to one* then every vertex in B has in-degree at most one. If the function f is *onto* then every vertex in B has in-degree at least one. If f is a bijection then every vertex in B has in-degree exactly equal to one.

Carl Pomerance's quote is taken from the home page of Doron Zeilberger.

Learning Objectives:

- Distinguish between *specification* and *implementation*, or equivalently between *mathematical functions* and *algorithms/programs*.
- Representing an object as a string (often of zeroes and ones).
- Examples of representations for common objects such as numbers, vectors, lists, and graphs.
- Prefix-free representations.
- Cantor's Theorem: The real numbers cannot be represented exactly as finite strings.

2
Computation and Representation

"The alphabet (sic) was a great invention, which enabled men (sic) to store and to learn with little effort what others had learned the hard way – that is, to learn from books rather than from direct, possibly painful, contact with the real world.", B.F. Skinner

"The name of the song is called 'HADDOCK'S EYES.'" [said the Knight]

"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed. *"That's what the name is CALLED. The name really is 'THE AGED AGED MAN.'"*

"Then I ought to have said 'That's what the SONG is called'?" Alice corrected herself.

"No, you oughtn't: that's quite another thing! The SONG is called 'WAYS AND MEANS': but that's only what it's CALLED, you know!"

"Well, what IS the song, then?" said Alice, who was by this time completely bewildered.

"I was coming to that," the Knight said. *"The song really IS 'A-SITTING ON A GATE': and the tune's my own invention."*

Lewis Carroll, *Through the Looking-Glass*

To a first approximation, *computation* is a process that maps an *input* to an *output*.

When discussing computation, it is essential to separate the question of **what** is the task we need to perform (i.e., the *specification*) from the question of **how** we achieve this task (i.e., the *implementation*). For example, as we've seen, there is more than one way to achieve the computational task of computing the product of two integers.

In this chapter we focus on the **what** part, namely defining computational tasks. For starters, we need to define the inputs and outputs. Capturing all the potential inputs and outputs that we might ever want to compute seems challenging, since computation today is applied to a wide variety of objects. We do not compute merely on numbers, but also on texts, images, videos, connection graphs of social

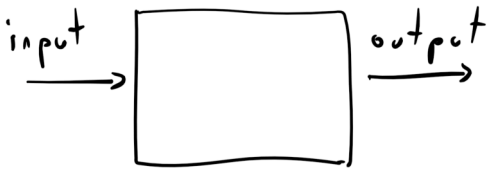


Figure 2.1: Our basic notion of *computation* is some process that maps an input to an output

networks, MRI scans, gene data, and even other programs. We will represent all these objects as **strings of zeroes and ones**, that is objects such as 0011101 or 1011 or any other finite list of 1's and 0's. (This choice is for convenience: there is nothing "holy" about zeroes and ones, and we could have used any other finite collection of symbols.)

Today, we are so used to the notion of digital representation that we are not surprised by the existence of such an encoding. But it is actually a deep insight with significant implications. Many animals can convey a particular fear or desire, but what is unique about humans is *language*: we use a finite collection of basic symbols to describe a potentially unlimited range of experiences. Language allows transmission of information over both time and space and enables societies that span a great many people and accumulate a body of shared knowledge over time.

Figure 2.2: We represent numbers, texts, images, networks and many other objects using strings of zeroes and ones. Writing the zeroes and ones themselves in green font over a black background is optional.

Over the last several decades, we have seen a revolution in what we can represent and convey in digital form. We can capture experiences with almost perfect fidelity, and disseminate it essentially instantaneously to an unlimited audience. Moreover, once information is in digital form, we can *compute* over it, and gain insights from data that were not accessible in prior times. At the heart of this revolution is the simple but profound observation that we can represent an unbounded variety of objects using a finite set of symbols (and in fact using only the two symbols `0` and `1`).

In later chapters, we will typically take such representations for granted, and hence use expressions such as "program P takes x as input" when x might be a number, a vector, a graph, or any other object, when we really mean that P takes as input the *representation* of x as a binary string. However, in this chapter we will dwell a bit more on how we can construct such representations.

This chapter: A non-mathy overview

The main takeaways from this chapter are:

- We can represent all kinds of objects we want to use as inputs and outputs using *binary strings*. For example, we can use the *binary basis* to represent integers and rational numbers as binary strings (see Section 2.1.1 and Section 2.2).
- We can *compose* the representations of simple objects to represent more complex objects. In this way, we can represent lists of integers or rational numbers, and use that to represent objects such as matrices, images, and graphs. *Prefix-free encoding* is one way to achieve such a composition (see Section 2.5.2).

- A *computational task* specifies a map from an input to an output— a *function*. It is crucially important to distinguish between the "what" and the "how", or the *specification* and *implementation* (see Section 2.6.1). A *function* simply defines which output corresponds to which input. It does not specify *how* to compute the output from the input, and as we've seen in the context of multiplication, there can be more than one way to compute the same function.
- While the set of all possible binary strings is infinite, it still cannot represent *everything*. In particular, there is no representation of the *real numbers* (with absolute accuracy) as binary strings. This result is also known as "Cantor's Theorem" (see Section 2.4) and is typically referred to as the result that the "reals are uncountable." It is also implied that there are *different levels* of infinity though we will not get into this topic in this book (see Remark 2.10).

The two "big ideas" we discuss are Big Idea 1 - we can compose representations for simple objects to represent more complex objects and Big Idea 2 - it is crucial to distinguish between *functions'* ("what") and *programs'* ("how"). The latter will be a theme we will come back to time and again in this book.

2.1 DEFINING REPRESENTATIONS

Every time we store numbers, images, sounds, databases, or other objects on a computer, what we actually store in the computer's memory is the *representation* of these objects. Moreover, the idea of representation is not restricted to digital computers. When we write down text or make a drawing we are *representing* ideas or experiences as sequences of symbols (which might as well be strings of zeroes and ones). Even our brain does not store the actual sensory inputs we experience, but rather only a *representation* of them.

To use objects such as numbers, images, graphs, or others as inputs for computation, we need to define precisely how to represent these objects as binary strings. A *representation scheme* is a way to map an object x to a binary string $E(x) \in \{0,1\}^*$. For example, a representation scheme for natural numbers is a function $E : \mathbb{N} \to \{0,1\}^*$. Of course, we cannot merely represent all numbers as the string "0011" (for example). A minimal requirement is that if two numbers x and x' are different then they would be represented by different strings. Another way to say this is that we require the encoding function E to be *one to one*.

2.1.1 Representing natural numbers

We now show how we can represent natural numbers as binary strings. Over the years people have represented numbers in a variety of ways, including Roman numerals, tally marks, our own Hindu-Arabic decimal system, and many others. We can use any one of those as well as many others to represent a number as a string (see Fig. 2.3). However, for the sake of concreteness, we use the *binary basis* as our default representation of natural numbers as strings. For example, we represent the number six as the string 110 since $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 6$, and similarly we represent the number thirty-five as the string $y = 100011$ which satisfies $\sum_{i=0}^{5} y_i \cdot 2^{|y|-i-1} = 35$. Some more examples are given in the table below.

Table 2.1: Representing numbers in the binary basis. The left-hand column contains representations of natural numbers in the decimal basis, while the right-hand column contains representations of the same numbers in the binary basis.

Number (decimal representation)	Number (binary representation)
0	0
1	1
2	10
5	101
16	10000
40	101000
53	110101
389	110000101
3750	111010100110

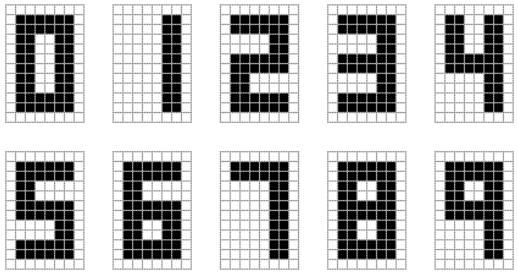


Figure 2.3: Representing each one the digits $0, 1, 2, \ldots, 9$ as a 12×8 bitmap image, which can be thought of as a string in $\{0,1\}^{96}$. Using this scheme we can represent a natural number x of n decimal digits as a string in $\{0,1\}^{96n}$. Image taken from blog post of A. C. Andersen.

If n is even, then the least significant digit of n's binary representation is 0, while if n is odd then this digit equals 1. Just like the number $\lfloor n/10 \rfloor$ corresponds to "chopping off" the least significant decimal digit (e.g., $\lfloor 457/10 \rfloor = \lfloor 45.7 \rfloor = 45$), the number $\lfloor n/2 \rfloor$ corresponds to the "chopping off" the least significant *binary* digit. Hence the binary representation can be formally defined as the following function $NtS : \mathbb{N} \to \{0,1\}^*$ (NtS stands for "natural numbers to strings"):

$$NtS(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ NtS(\lfloor n/2 \rfloor) parity(n) & n > 1 \end{cases} \tag{2.1}$$

where $parity : \mathbb{N} \to \{0,1\}$ is the function defined as $parity(n) = 0$ if n is even and $parity(n) = 1$ if n is odd, and as usual, for strings $x, y \in \{0,1\}^*$, xy denotes the concatenation of x and y. The function

NtS is defined *recursively*: for every $n > 1$ we define $rep(n)$ in terms of the representation of the smaller number $\lfloor n/2 \rfloor$. It is also possible to define NtS non-recursively, see Exercise 2.2.

Throughout most of this book, the particular choices of representation of numbers as binary strings would not matter much: we just need to know that such a representation exists. In fact, for many of our purposes we can even use the simpler representation of mapping a natural number n to the length-n all-zero string 0^n.

Remark 2.1 — Binary representation in python (optional). We can implement the binary representation in *Python* as follows:

```
def NtS(n):# natural numbers to strings
    if n > 1:
        return NtS(n // 2) + str(n % 2)
    else:
        return str(n % 2)

print(NtS(236))
# 11101100

print(NtS(19))
# 10011
```

We can also use Python to implement the inverse transformation, mapping a string back to the natural number it represents.

```
def StN(x):# String to number
    k = len(x)-1
    return sum(int(x[i])*(2**(k-i)) for i in
     ↪  range(k+1))

print(StN(NtS(236)))
# 236
```


Remark 2.2 — Programming examples. In this book, we sometimes use *code examples* as in Remark 2.1. The point is always to emphasize that certain computations can be achieved concretely, rather than illustrating the features of Python or any other programming language. Indeed, one of the messages of this book is that all programming languages are in a certain precise sense *equivalent* to one another, and hence we could have just as well used JavaScript, C, COBOL, Visual Basic or even BrainF*ck. This book is *not* about programming, and it is absolutely OK if

you are not familiar with Python or do not follow code examples such as those in Remark 2.1.

2.1.2 Meaning of representations (discussion)

It is natural for us to think of 236 as the "actual" number, and of 11101100 as "merely" its representation. However, for most Europeans in the middle ages CCXXXVI would be the "actual" number and 236 (if they have even heard about it) would be the weird Hindu-Arabic positional representation.[1] When our AI robot overlords materialize, they will probably think of 11101100 as the "actual" number and of 236 as "merely" a representation that they need to use when they give commands to humans.

So what is the "actual" number? This is a question that philosophers of mathematics have pondered throughout history. Plato argued that mathematical objects exist in some ideal sphere of existence (that to a certain extent is more "real" than the world we perceive via our senses, as this latter world is merely the shadow of this ideal sphere). In Plato's vision, the symbols 236 are merely notation for some ideal object, that, in homage to the late musician, we can refer to as "the number commonly represented by 236".

The Austrian philosopher Ludwig Wittgenstein, on the other hand, argued that mathematical objects do not exist at all, and the only things that exist are the actual marks on paper that make up 236, 11101100 or CCXXXVI. In Wittgenstein's view, mathematics is merely about formal manipulation of symbols that do not have any inherent meaning. You can think of the "actual" number as (somewhat recursively) "that thing which is common to 236, 11101100 and CCXXXVI and all other past and future representations that are meant to capture the same object".

While reading this book, you are free to choose your own philosophy of mathematics, as long as you maintain the distinction between the mathematical objects themselves and the various particular choices of representing them, whether as splotches of ink, pixels on a screen, zeroes and ones, or any other form.

2.2 REPRESENTATIONS BEYOND NATURAL NUMBERS

We have seen that natural numbers can be represented as binary strings. We now show that the same is true for other types of objects, including (potentially negative) integers, rational numbers, vectors, lists, graphs and many others. In many instances, choosing the "right" string representation for a piece of data is highly non-trivial, and finding the "best" one (e.g., most compact, best fidelity, most efficiently manipulable, robust to errors, most informative features, etc.) is the

[1] While the Babylonians already invented a positional system much earlier, the decimal positional system we use today was invented by Indian mathematicians around the third century. Arab mathematicians took it up in the 8th century. It first received significant attention in Europe with the publication of the 1202 book *"Liber Abaci"* by Leonardo of Pisa, also known as Fibonacci, but it did not displace Roman numerals in common usage until the 15th century.

object of intense research. But for now, we focus on presenting some simple representations for various objects that we would like to use as inputs and outputs for computation.

2.2.1 Representing (potentially negative) integers

Since we can represent natural numbers as strings, we can represent the full set of *integers* (i.e., members of the set $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, +1, +2, +3, \ldots\}$) by adding one more bit that represents the sign. To represent a (potentially negative) number m, we prepend to the representation of the natural number $|m|$ a bit σ that equals 0 if $m \geq 0$ and equals 1 if $m < 0$. Formally, we define the function $ZtS : \mathbb{Z} \to \{0,1\}^*$ as follows

$$ZtS(m) = \begin{cases} 0\ NtS(m) & m \geq 0 \\ 1\ NtS(-m) & m < 0 \end{cases}$$

where NtS is defined as in (2.1).

While the encoding function of a representation needs to be one to one, it does not have to be *onto*. For example, in the representation above there is no number that is represented by the empty string but it is still a fine representation, since every integer is represented uniquely by some string.

R

Remark 2.3 — Interpretation and context. Given a string $y \in \{0,1\}^*$, how do we know if it's "supposed" to represent a (non-negative) natural number or a (potentially negative) integer? For that matter, even if we know y is "supposed" to be an integer, how do we know what representation scheme it uses? The short answer is that we do not necessarily know this information, unless it is supplied from the context. (In programming languages, the compiler or interpreter determines the representation of the sequence of bits corresponding to a variable based on the variable's *type*.) We can treat the same string y as representing a natural number, an integer, a piece of text, an image, or a green gremlin. Whenever we say a sentence such as "let n be the number represented by the string y," we will assume that we are fixing some canonical representation scheme such as the ones above. The choice of the particular representation scheme will rarely matter, except that we want to make sure to stick with the same one for consistency.

2.2.2 Two's complement representation (optional)

Section 2.2.1's approach of representing an integer using a specific "sign bit" is known as the *Signed Magnitude Representation* and was

used in some early computers. However, the two's complement representation is much more common in practice. The *two's complement representation* of an integer k in the set $\{-2^n, -2^n+1, \ldots, 2^n-1\}$ is the string $ZtS_n(k)$ of length $n+1$ defined as follows:

$$ZtS_n(k) = \begin{cases} NtS_{n+1}(k) & 0 \leq k \leq 2^n - 1 \\ NtS_{n+1}(2^{n+1}+k) & -2^n \leq k \leq -1 \end{cases},$$

where $NtS_\ell(m)$ denotes the standard binary representation of a number $m \in \{0, \ldots, 2^\ell\}$ as string of length ℓ, padded with leading zeros as needed. For example, if $n = 3$ then $ZtS_3(1) = NtS_4(1) = 0001$, $ZtS_3(2) = NtS_4(2) = 0010$, $ZtS_3(-1) = NtS_4(16-1) = 1111$, and $ZtS_3(-8) = NtS_4(16-8) = 1000$. If k is a negative number larger than or equal to -2^n then $2^{n+1}+k$ is a number between 2^n and $2^{n+1}-1$. Hence the two's complement representation of such a number k is a string of length $n+1$ with its first digit equal to 1.

Another way to say this is that we represent a potentially negative number $k \in \{-2^n, \ldots, 2^n-1\}$ as the non-negative number $k \mod 2^{n+1}$ (see also Fig. 2.4). This means that if two (potentially negative) numbers k and k' are not too large (i.e., $|k| + |k'| < 2^{n+1}$), then we can compute the representation of $k+k'$ by adding modulo 2^{n+1} the representations of k and k' as if they were non-negative integers. This property of the two's complement representation is its main attraction since, depending on their architectures, microprocessors can often perform arithmetic operations modulo 2^w very efficiently (for certain values of w such as 32 and 64). Many systems leave it to the programmer to check that values are not too large and will carry out this modular arithmetic regardless of the size of the numbers involved. For this reason, in some systems adding two large positive numbers can result in a *negative* number (e.g., adding $2^n - 100$ and $2^n - 200$ might result in -300 since $(2^{n+1} - 300) \mod 2^{n+1} = -300$, see also Fig. 2.4).

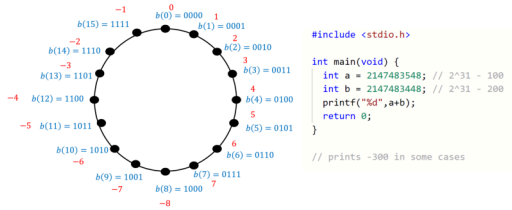


Figure 2.4: In the *two's complement representation* we represent a potentially negative integer $k \in \{-2^n, \ldots, 2^n-1\}$ as an $n+1$ length string using the binary representation of the integer $k \mod 2^{n+1}$. On the left-hand side: this representation for $n = 3$ (the red integers are the numbers being represented by the blue binary strings). If a microprocessor does not check for overflows, adding the two positive numbers 6 and 5 might result in the negative number -5 (since $-5 \mod 16 = 11$. The right-hand side is a `C` program that will on some 32 bit architecture print a negative number after adding two positive numbers. (Integer overflow in `C` is considered *undefined behavior* which means the result of this program, including whether it runs or crashes, could differ depending on the architecture, compiler, and even compiler options and version.)

2.2.3 Rational numbers and representing pairs of strings

We can represent a rational number of the form a/b by representing the two numbers a and b. However, merely concatenating the representations of a and b will not work. For example, the binary representation of 4 is 100 and the binary representation of 43 is 101011, but the concatenation 100101011 of these strings is also the concatenation of the representation 10010 of 18 and the representation 1011 of 11. Hence, if we used such simple concatenation then we would not be able to tell if the string 100101011 is supposed to represent $4/43$ or $18/11$.

We tackle this by giving a general representation for *pairs of strings*. If we were using a pen and paper, we would just use a separator symbol such as $\|$ to represent, for example, the pair consisting of the num-

bers represented by 10 and 110001 as the length-9 string "10‖110001". In other words, there is a one to one map F from *pairs of strings* $x, y \in \{0,1\}^*$ into a single string z over the alphabet $\Sigma = \{0,1,\|\}$ (in other words, $z \in \Sigma^*$). Using such separators is similar to the way we use spaces and punctuation to separate words in English. By adding a little redundancy, we achieve the same effect in the digital domain. We can map the three-element set Σ to the three-element set $\{00,11,01\} \subset \{0,1\}^2$ in a one-to-one fashion, and hence encode a length n string $z \in \Sigma^*$ as a length $2n$ string $w \in \{0,1\}^*$.

Our final representation for rational numbers is obtained by composing the following steps:

1. Representing a non-negative rational number as a pair of natural numbers.
2. Representing a natural number by a string via the binary representation.
3. Combining 1 and 2 to obtain a representation of a rational number as a pair of strings.
4. Representing a pair of strings over $\{0,1\}$ as a single string over $\Sigma = \{0,1,\|\}$.
5. Representing a string over Σ as a longer string over $\{0,1\}$.

■ **Example 2.4 — Representing a rational number as a string.** Consider the rational number $r = -5/8$. We represent -5 as 1101 and $+8$ as 01000, and so we can represent r as the *pair* of strings $(1101, 01000)$ and represent this pair as the length 10 string $1101\|01000$ over the alphabet $\{0,1,\|\}$. Now, applying the map $0 \mapsto 00$, $1 \mapsto 11$, $\| \mapsto 01$, we can represent the latter string as the length 20 string $s = 11110011010011000000$ over the alphabet $\{0,1\}$.

The same idea can be used to represent triples of strings, quadruples, and so on as a string. Indeed, this is one instance of a very general principle that we use time and again in both the theory and practice of computer science (for example, in Object Oriented programming):

Big Idea 1 If we can represent objects of type T as strings, then we can represent tuples of objects of type T as strings as well.

Repeating the same idea, once we can represent objects of type T, we can also represent *lists of lists* of such objects, and even lists of lists of lists and so on and so forth. We will come back to this point when we discuss *prefix free encoding* in Section 2.5.2.

2.3 REPRESENTING REAL NUMBERS

The set of *real numbers* $\mathbb{R}$ contains all numbers including positive, negative, and fractional, as well as *irrational* numbers such as π or e. Every real number can be approximated by a rational number, and thus we can represent every real number x by a rational number a/b that is very close to x. For example, we can represent π by $22/7$ within an error of about 10^{-3}. If we want a smaller error (e.g., about 10^{-4}) then we can use $311/99$, and so on and so forth.

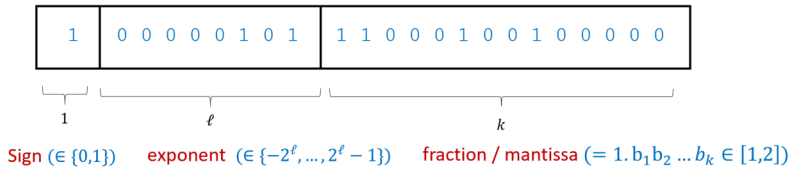


Figure 2.5: The *floating-point representation* of a real number $x \in \mathbb{R}$ is its approximation as a number of the form $\sigma b \cdot 2^e$ where $\sigma \in \{\pm 1\}$, e is an (potentially negative) integer, and b is a rational number between 1 and 2 expressed as a binary fraction $1.b_1 b_2 \ldots b_k$ for some $b_1, \ldots, b_k \in \{0,1\}$ (that is $b = 1 + b_1/2 + b_2/4 + \ldots + b_k/2^k$). Commonly-used floating-point representations fix the numbers ℓ and k of bits to represent e and b respectively. In the example above, assuming we use two's complement representation for e, the number represented is $-1 \times 2^5 \times (1 + 1/2 + 1/4 + 1/64 + 1/512) = -56.5625$.

The above representation of real numbers via rational numbers that approximate them is a fine choice for a representation scheme. However, typically in computing applications, it is more common to use the *floating-point representation scheme* (see Fig. 2.5) to represent real numbers. In the floating-point representation scheme we represent $x \in \mathbb{R}$ by the pair (b, e) of (positive or negative) integers of some prescribed sizes (determined by the desired accuracy) such that $b \times 2^e$ is closest to x. Floating-point representation is the base-two version of scientific notation, where one represents a number $y \in R$ as its approximation of the form $b \times 10^e$ for b, e. It is called "floating-point" because we can think of the number b as specifying a sequence of binary digits, and e as describing the location of the "binary point" within this sequence. The use of floating representation is the reason why in many programming systems, printing the expression `0.1+0.2` will result in `0.30000000000000004` and not `0.3`, see here, here and here for more.

The reader might be (rightly) worried about the fact that the floating-point representation (or the rational number one) can only *approximately* represent real numbers. In many (though not all) computational applications, one can make the accuracy tight enough so that this does not affect the final result, though sometimes we do need to be careful. Indeed, floating-point bugs can sometimes be no joking matter. For example, floating-point rounding errors have been implicated in the failure of a U.S. Patriot missile to intercept an Iraqi Scud missile, costing 28 lives, as well as a 100 million pound error in computing payouts to British pensioners.

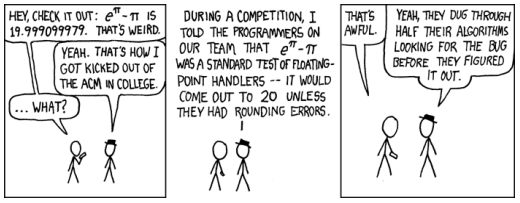


Figure 2.6: XKCD cartoon on floating-point arithmetic.

2.4 CANTOR'S THEOREM, COUNTABLE SETS, AND STRING REPRESENTATIONS OF THE REAL NUMBERS

"For any collection of fruits, we can make more fruit salads than there are fruits. If not, we could label each salad with a different fruit, and consider the salad of all fruits not in their salad. The label of this salad is in it if and only if it is not.", Martha Storey.

Given the issues with floating-point approximations for real numbers, a natural question is whether it is possible to represent real numbers *exactly* as strings. Unfortunately, the following theorem shows that this cannot be done:

Theorem 2.5 — Cantor's Theorem. There does not exist a one-to-one function $RtS : \mathbb{R} \to \{0,1\}^*$. [2]

[2] RtS stands for "real numbers to strings".

Countable sets. We say that a set S is *countable* if there is an onto map $C : \mathbb{N} \to S$, or in other words, we can write S as the sequence $C(0), C(1), C(2), \ldots$. Since the binary representation yields an onto map from $\{0,1\}^*$ to $\mathbb{N}$, and the composition of two onto maps is onto, a set S is countable iff there is an onto map from $\{0,1\}^*$ to S. Using the basic properties of functions (see Section 1.4.3), a set is countable if and only if there is a one-to-one function from S to $\{0,1\}^*$. Hence, we can rephrase Theorem 2.5 as follows:

Theorem 2.6 — Cantor's Theorem (equivalent statement). The reals are uncountable. That is, there does not exist an onto function $NtR : \mathbb{N} \to \mathbb{R}$.

Theorem 2.6 was proven by Georg Cantor in 1874. This result (and the theory around it) was quite shocking to mathematicians at the time. By showing that there is no one-to-one map from $\mathbb{R}$ to $\{0,1\}^*$ (or $\mathbb{N}$), Cantor showed that these two infinite sets have "different forms of infinity" and that the set of real numbers $\mathbb{R}$ is in some sense "bigger" than the infinite set $\{0,1\}^*$. The notion that there are "shades of infinity" was deeply disturbing to mathematicians and philosophers at the time. The philosopher Ludwig Wittgenstein (whom we mentioned before) called Cantor's results "utter nonsense" and "laughable." Others thought they were even worse than that. Leopold Kronecker called Cantor a "corrupter of youth," while Henri Poincaré said that Cantor's ideas "should be banished from mathematics once and for all." The tide eventually turned, and these days Cantor's work is universally accepted as the cornerstone of set theory and the foundations of mathematics. As David Hilbert said in 1925, *"No one shall expel us from the paradise which Cantor has created for us."* As we will see later in this

book, Cantor's ideas also play a huge role in the theory of computation.

Now that we have discussed Theorem 2.5's importance, let us see the proof. It is achieved in two steps:

1. Define some infinite set $\mathcal{X}$ for which it is easier for us to prove that $\mathcal{X}$ is not countable (namely, it's easier for us to prove that there is no one-to-one function from $\mathcal{X}$ to $\{0,1\}^*$).

2. Prove that there *is* a one-to-one function G mapping $\mathcal{X}$ to $\mathbb{R}$.

We can use a proof by contradiction to show that these two facts together imply Theorem 2.5. Specifically, if we assume (towards the sake of contradiction) that there exists some one-to-one F mapping $\mathbb{R}$ to $\{0,1\}^*$, then the function $x \mapsto F(G(x))$ obtained by composing F with the function G from Step 2 above would be a one-to-one function from $\mathcal{X}$ to $\{0,1\}^*$, which contradicts what we proved in Step 1!

To turn this idea into a full proof of Theorem 2.5 we need to:

- Define the set $\mathcal{X}$.
- Prove that there is no one-to-one function from $\mathcal{X}$ to $\{0,1\}^*$
- Prove that there *is* a one-to-one function from $\mathcal{X}$ to $\mathbb{R}$.

We now proceed to do precisely that. That is, we will define the set $\{0,1\}^\infty$, which will play the role of $\mathcal{X}$, and then state and prove two lemmas that show that this set satisfies our two desired properties.

Definition 2.7 We denote by $\{0,1\}^\infty$ the set $\{f \mid f : \mathbb{N} \to \{0,1\}\}$.

That is, $\{0,1\}^\infty$ is a set of *functions*, and a function f is in $\{0,1\}^\infty$ iff its domain is $\mathbb{N}$ and its codomain is $\{0,1\}$. We can also think of $\{0,1\}^\infty$ as the set of all infinite *sequences* of bits, since a function $f : \mathbb{N} \to \{0,1\}$ can be identified with the sequence $(f(0), f(1), f(2), ...)$. The following two lemmas show that $\{0,1\}^\infty$ can play the role of $\mathcal{X}$ to establish Theorem 2.5.

Lemma 2.8 There does not exist a one-to-one map $FtS : \{0,1\}^\infty \to \{0,1\}^*$.[3]

Lemma 2.9 There *does* exist a one-to-one map $FtR : \{0,1\}^\infty \to \mathbb{R}$.[4]

As we've seen above, Lemma 2.8 and Lemma 2.9 together imply Theorem 2.5. To repeat the argument more formally, suppose, for the sake of contradiction, that there did exist a one-to-one function $RtS : \mathbb{R} \to \{0,1\}^*$. By Lemma 2.9, there exists a one-to-one function $FtR : \{0,1\}^\infty \to \mathbb{R}$. Thus, under this assumption, since the composition of two one-to-one functions is one-to-one (see Exercise 2.12), the

[3] FtS stands for "functions to strings".

[4] FtR stands for "functions to reals."

function $FtS : \{0,1\}^\infty \to \{0,1\}^*$ defined as $FtS(f) = RtS(FtR(f))$ will be one to one, contradicting Lemma 2.8. See Fig. 2.7 for a graphical illustration of this argument.

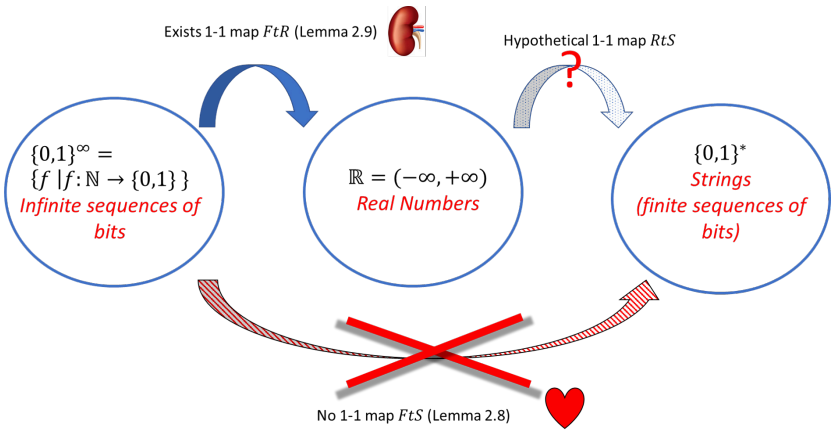
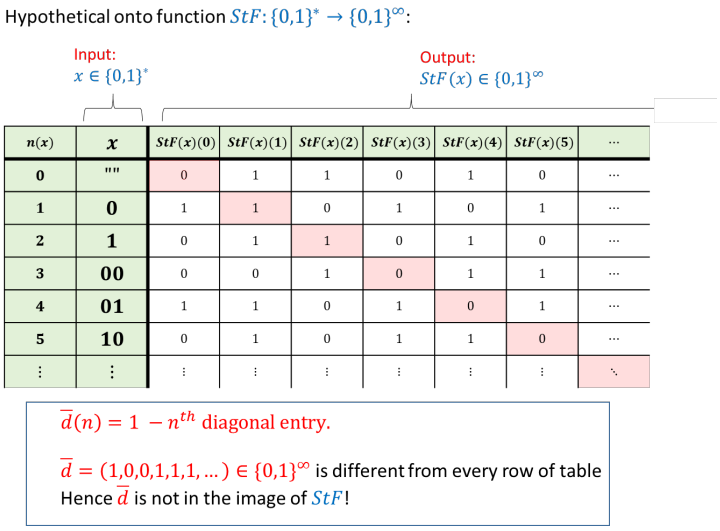


Figure 2.7: We prove Theorem 2.5 by combining Lemma 2.8 and Lemma 2.9. Lemma 2.9, which uses standard calculus tools, shows the existence of a one-to-one map FtR from the set $\{0,1\}^\infty$ to the real numbers. So, if a hypothetical one-to-one map $RtS : \mathbb{R} \to \{0,1\}^*$ existed, then we could compose them to get a one-to-one map $FtS : \{0,1\}^\infty \to \{0,1\}^*$. Yet this contradicts Lemma 2.8- the heart of the proof- which rules out the existence of such a map.

Now all that is left is to prove these two lemmas. We start by proving Lemma 2.8 which is really the heart of Theorem 2.5.



$n(x)$	x	$StF(x)(0)$	$StF(x)(1)$	$StF(x)(2)$	$StF(x)(3)$	$StF(x)(4)$	$StF(x)(5)$	⋯
0	""	0	1	1	0	1	0	⋯
1	0	1	1	0	1	0	1	⋯
2	1	0	1	1	0	1	0	⋯
3	00	0	0	1	0	1	1	⋯
4	01	1	1	0	1	0	1	⋯
5	10	0	1	0	1	1	0	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Figure 2.8: We construct a function $\overline{d}$ such that $\overline{d} \neq StF(x)$ for every $x \in \{0,1\}^*$ by ensuring that $\overline{d}(n(x)) \neq StF(x)(n(x))$ for every $x \in \{0,1\}^*$ with lexicographic order $n(x)$. We can think of this as building a table where the columns correspond to numbers $m \in \mathbb{N}$ and the rows correspond to $x \in \{0,1\}^*$ (sorted according to $n(x)$). If the entry in the x-th row and the m-th column corresponds to $g(m))$ where $g = StF(x)$ then $\overline{d}$ is obtained by going over the "diagonal" elements in this table (the entries corresponding to the x-th row and $n(x)$-th column) and ensuring that $\overline{d}(x)(n(x)) \neq StF(x)(x)$.

Warm-up: "Baby Cantor". The proof of Lemma 2.8 is rather subtle. One way to get intuition for it is to consider the following finite statement "there is no onto function $f : \{0,\ldots,99\} \to \{0,1\}^{100}$". Of course we know it's true since the set $\{0,1\}^{100}$ is bigger than the set $[100]$, but let's see a direct proof. For every $f : \{0,\ldots,99\} \to \{0,1\}^{100}$, we can define the string $\overline{d} \in \{0,1\}^{100}$ as follows: $\overline{d} = (1 - f(0)_0, 1 - f(1)_1, \ldots, 1 - f(99)_{99})$. If f was onto, then there would exist some $n \in [100]$ such that $f(n) = \overline{d}$, but we claim that no such n exists.

Indeed, if there was such n, then the n-th coordinate of $\overline{d}$ would equal $f(n)_n$ but by definition this coordinate equals $1 - f(n)_n$. See also a "proof by code" of this statement.

Proof of Lemma 2.8. We will prove that there does not exist an *onto* function $StF : \{0,1\}^* \rightarrow \{0,1\}^\infty$. This implies the lemma since for every two sets A and B, there exists an onto function from A to B if and only if there exists a one-to-one function from B to A (see Lemma 1.2).

The technique of this proof is known as the "diagonal argument" and is illustrated in Fig. 2.8. We assume, towards a contradiction, that there exists such a function $StF : \{0,1\}^* \rightarrow \{0,1\}^\infty$. We will show that StF is not onto by demonstrating a function $\overline{d} \in \{0,1\}^\infty$ such that $\overline{d} \neq StF(x)$ for every $x \in \{0,1\}^*$. Consider the lexicographic ordering of binary strings (i.e., "",0,1,00,01,...). For every $n \in \mathbb{N}$, we let x_n be the n-th string in this order. That is $x_0 =$ "", $x_1 = 0$, $x_2 = 1$ and so on and so forth. We define the function $\overline{d} \in \{0,1\}^\infty$ as follows:

$$\overline{d}(n) = 1 - StF(x_n)(n)$$

for every $n \in \mathbb{N}$. That is, to compute $\overline{d}$ on input $n \in \mathbb{N}$, we first compute $g = StF(x_n)$, where $x_n \in \{0,1\}^*$ is the n-th string in the lexicographical ordering. Since $g \in \{0,1\}^\infty$, it is a function mapping $\mathbb{N}$ to $\{0,1\}$. The value $\overline{d}(n)$ is defined to be the negation of $g(n)$.

The definition of the function $\overline{d}$ is a bit subtle. One way to think about it is to imagine the function StF as being specified by an infinitely long table, in which every row corresponds to a string $x \in \{0,1\}^*$ (with strings sorted in lexicographic order), and contains the sequence $StF(x)(0), StF(x)(1), StF(x)(2), \ldots$. The *diagonal* elements in this table are the values

$$StF("")(0), StF(0)(1), StF(1)(2), StF(00)(3), StF(01)(4), \ldots$$

which correspond to the elements $StF(x_n)(n)$ in the n-th row and n-th column of this table for $n = 0, 1, 2, \ldots$. The function $\overline{d}$ we defined above maps every $n \in \mathbb{N}$ to the negation of the n-th diagonal value.

To complete the proof that StF is not onto we need to show that $\overline{d} \neq StF(x)$ for every $x \in \{0,1\}^*$. Indeed, let $x \in \{0,1\}^*$ be some string and let $g = StF(x)$. If n is the position of x in the lexicographical order then by construction $\overline{d}(n) = 1 - g(n) \neq g(n)$ which means that $g \neq \overline{d}$ which is what we wanted to prove.

■

R

Remark 2.10 — Generalizing beyond strings and reals. Lemma 2.8 doesn't really have much to do with the natural numbers or the strings. An examination of the proof shows that it really shows that for *every* set S, there is no one-to-one map $F : \{0,1\}^S \to S$ where $\{0,1\}^S$ denotes the set $\{f \mid f : S \to \{0,1\}\}$ of all Boolean functions with domain S. Since we can identify a subset $V \subseteq S$ with its characteristic function $f = 1_V$ (i.e., $1_V(x) = 1$ iff $x \in V$), we can think of $\{0,1\}^S$ also as the set of all *subsets* of S. This subset is sometimes called the *power set* of S and denoted by $\mathcal{P}(S)$ or 2^S.

The proof of Lemma 2.8 can be generalized to show that there is no one-to-one map between a set and its power set. In particular, it means that the set $\{0,1\}^{\mathbb{R}}$ is "even bigger" than $\mathbb{R}$. Cantor used these ideas to construct an infinite hierarchy of shades of infinity. The number of such shades turns out to be much larger than $|\mathbb{N}|$ or even $|\mathbb{R}|$. He denoted the cardinality of $\mathbb{N}$ by $\aleph_0$ and denoted the next largest infinite number by $\aleph_1$. ($\aleph$ is the first letter in the Hebrew alphabet.) Cantor also made the continuum hypothesis that $|\mathbb{R}| = \aleph_1$. We will come back to the fascinating story of this hypothesis later on in this book. This lecture of Aaronson mentions some of these issues (see also this Berkeley CS 70 lecture).

To complete the proof of Theorem 2.5, we need to show Lemma 2.9. This requires some calculus background but is otherwise straightforward. If you have not had much experience with limits of a real series before, then the formal proof below might be a little hard to follow. This part is not the core of Cantor's argument, nor are such limits important to the remainder of this book, so you can feel free to take Lemma 2.9 on faith and skip the proof.

Proof Idea:

We define $FtR(f)$ to be the number between 0 and 2 whose decimal expansion is $f(0).f(1)f(2)\ldots$, or in other words $FtR(f) = \sum_{i=0}^{\infty} f(i) \cdot 10^{-i}$. If f and g are two distinct functions in $\{0,1\}^{\infty}$, then there must be some input k in which they disagree. If we take the minimum such k, then the numbers $f(0).f(1)f(2)\ldots f(k-1)f(k)\ldots$ and $g(0).g(1)g(2)\ldots g(k)\ldots$ agree with each other all the way up to the $k-1$-th digit after the decimal point, and disagree on the k-th digit. But then these numbers must be distinct. Concretely, if $f(k) = 1$ and $g(k) = 0$ then the first number is larger than the second, and otherwise ($f(k) = 0$ and $g(k) = 1$) the first number is smaller than the second. In the proof we have to be a little careful since these are numbers with *infinite expansions*. For example, the number one half has two decimal

expansions 0.5 and $0.49999\cdots$. However, this issue does not come up here, since we restrict attention only to numbers with decimal expansions that do not involve the digit 9.

⋆

Proof of Lemma 2.9. For every $f \in \{0,1\}^\infty$, we define $FtR(f)$ to be the number whose decimal expansion is $f(0).f(1)f(2)f(3)\ldots$. Formally,

$$FtR(f) = \sum_{i=0}^{\infty} f(i) \cdot 10^{-i} \tag{2.2}$$

It is a known result in calculus (whose proof we will not repeat here) that the series on the right-hand side of (2.2) converges to a definite limit in $\mathbb{R}$.

We now prove that FtR is one to one. Let f, g be two distinct functions in $\{0,1\}^\infty$. Since f and g are distinct, there must be some input on which they differ, and we define k to be the smallest such input and assume without loss of generality that $f(k) = 0$ and $g(k) = 1$. (Otherwise, if $f(k) = 1$ and $g(k) = 0$, then we can simply switch the roles of f and g.) The numbers $FtR(f)$ and $FtR(g)$ agree with each other up to the $k-1$-th digit up after the decimal point. Since this digit equals 0 for $FtR(f)$ and equals 1 for $FtR(g)$, we claim that $FtR(g)$ is bigger than $FtR(f)$ by at least $0.5 \cdot 10^{-k}$. To see this note that the difference $FtR(g) - FtR(f)$ will be minimized if $g(\ell) = 0$ for every $\ell > k$ and $f(\ell) = 1$ for every $\ell > k$, in which case (since f and g agree up to the $k-1$-th digit)

$$FtR(g) - FtR(f) = 10^{-k} - 10^{-k-1} - 10^{-k-2} - 10^{-k-3} - \cdots \tag{2.3}$$

Since the infinite series $\sum_{i=0}^{\infty} 10^{-i}$ converges to $10/9$, it follows that for every such f and g, $FtR(g) - FtR(f) \geq 10^{-k} - 10^{-k-1} \cdot (10/9) > 0$. In particular we see that for every distinct $f, g \in \{0,1\}^\infty$, $FtR(f) \neq FtR(g)$, implying that the function FtR is one to one.

■

R

Remark 2.11 — Using decimal expansion (optional). In the proof above we used the fact that $1 + 1/10 + 1/100 + \cdots$ converges to $10/9$, which plugging into (2.3) yields that the difference between $FtR(g)$ and $FtR(h)$ is at least $10^{-k} - 10^{-k-1} \cdot (10/9) > 0$. While the choice of the decimal representation for FtR was arbitrary, we could not have used the binary representation in its place. Had we used the *binary* expansion instead of decimal, the corresponding sequence $1 + 1/2 + 1/4 + \cdots$ converges to $2/1 = 2$,

and since $2^{-k} = 2^{-k-1} \cdot 2$, we could not have deduced that FtR is one to one. Indeed there do exist pairs of distinct sequences $f, g \in \{0,1\}^\infty$ such that $\sum_{i=0}^{\infty} f(i)2^{-i} = \sum_{i=0}^{\infty} g(i)2^{-i}$. (For example, the sequence $1,0,0,0,...$ and the sequence $0,1,1,1,...$ have this property.)

2.4.1 Corollary: Boolean functions are uncountable

Cantor's Theorem yields the following corollary that we will use several times in this book: the set of all *Boolean functions* (mapping $\{0,1\}^*$ to $\{0,1\}$) is not countable:

Theorem 2.12 — Boolean functions are uncountable. Let ALL be the set of all functions $F : \{0,1\}^* \to \{0,1\}$. Then ALL is uncountable. Equivalently, there does not exist an onto map $StALL : \{0,1\}^* \to ALL$.

Proof Idea:

This is a direct consequence of Lemma 2.8, since we can use the binary representation to show a one-to-one map from $\{0,1\}^\infty$ to ALL. Hence the uncountability of $\{0,1\}^\infty$ implies the uncountability of ALL.

⋆

Proof of Theorem 2.12. Since $\{0,1\}^\infty$ is uncountable, the result will follow by showing a one-to-one map from $\{0,1\}^\infty$ to ALL. The reason is that the existence of such a map implies that if ALL was countable, and hence there was a one-to-one map from ALL to $\mathbb{N}$, then there would have been a one-to-one map from $\{0,1\}^\infty$ to $\mathbb{N}$, contradicting Lemma 2.8.

We now show this one-to-one map. We simply map a function $f \in \{0,1\}^\infty$ to the function $F : \{0,1\}^* \to \{0,1\}$ as follows. We let $F(0) = f(0)$, $F(1) = f(1)$, $F(10) = f(2)$, $F(11) = f(3)$ and so on and so forth. That is, for every $x \in \{0,1\}^*$ that represents a natural number n in the binary basis, we define $F(x) = f(n)$. If x does not represent such a number (e.g., it has a leading zero), then we set $F(x) = 0$.

This map is one-to-one since if $f \neq g$ are two distinct elements in $\{0,1\}^\infty$, then there must be some input $n \in \mathbb{N}$ on which $f(n) \neq g(n)$. But then if $x \in \{0,1\}^*$ is the string representing n, we see that $F(x) \neq G(x)$ where F is the function in ALL that f mapped to, and G is the function that g is mapped to.

■

2.4.2 Equivalent conditions for countability

The results above establish many equivalent ways to phrase the fact that a set is countable. Specifically, the following statements are all equivalent:

1. The set S is countable
2. There exists an onto map from $\mathbb{N}$ to S
3. There exists an onto map from $\{0,1\}^*$ to S.
4. There exists a one-to-one map from S to $\mathbb{N}$
5. There exists a one-to-one map from S to $\{0,1\}^*$.
6. There exists an onto map from some countable set T to S.
7. There exists a one-to-one map from S to some countable set T.

Make sure you know how to prove the equivalence of all the results above.

2.5 REPRESENTING OBJECTS BEYOND NUMBERS

Numbers are of course by no means the only objects that we can represent as binary strings. A *representation scheme* for representing objects from some set $\mathcal{O}$ consists of an *encoding* function that maps an object in $\mathcal{O}$ to a string, and a *decoding* function that decodes a string back to an object in $\mathcal{O}$. Formally, we make the following definition:

Definition 2.13 — String representation. Let $\mathcal{O}$ be any set. A *representation scheme* for $\mathcal{O}$ is a pair of functions E, D where $E : \mathcal{O} \to \{0,1\}^*$ is a total one-to-one function, $D : \{0,1\}^* \to_p \mathcal{O}$ is a (possibly partial) function, and such that D and E satisfy that $D(E(o)) = o$ for every $o \in \mathcal{O}$. E is known as the *encoding* function and D is known as the *decoding* function.

Note that the condition $D(E(o)) = o$ for every $o \in \mathcal{O}$ implies that D is *onto* (can you see why?). It turns out that to construct a representation scheme we only need to find an *encoding* function. That is, every one-to-one encoding function has a corresponding decoding function, as shown in the following lemma:

Lemma 2.14 Suppose that $E : \mathcal{O} \to \{0,1\}^*$ is one-to-one. Then there exists a function $D : \{0,1\}^* \to \mathcal{O}$ such that $D(E(o)) = o$ for every $o \in \mathcal{O}$.

Proof. Let o_0 be some arbitrary element of $\mathcal{O}$. For every $x \in \{0,1\}^*$, there exists either zero or a single $o \in \mathcal{O}$ such that $E(o) = x$ (otherwise E would not be one-to-one). We will define $D(x)$ to equal o_0 in the first case and this single object o in the second case. By definition $D(E(o)) = o$ for every $o \in \mathcal{O}$.

■

Remark 2.15 — Total decoding functions. While the decoding function of a representation scheme can in general be a *partial* function, the proof of Lemma 2.14 implies that every representation scheme has a *total* decoding function. This observation can sometimes be useful.

2.5.1 Finite representations

If $\mathcal{O}$ is *finite*, then we can represent every object in $\mathcal{O}$ as a string of length at most some number n. What is the value of n? Let us denote by $\{0,1\}^{\leq n}$ the set $\{x \in \{0,1\}^* : |x| \leq n\}$ of strings of length at most n. The size of $\{0,1\}^{\leq n}$ is equal to

$$|\{0,1\}^0| + |\{0,1\}^1| + |\{0,1\}^2| + \cdots + |\{0,1\}^n| = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

using the standard formula for summing a geometric progression.

To obtain a representation of objects in $\mathcal{O}$ as strings of length at most n we need to come up with a one-to-one function from $\mathcal{O}$ to $\{0,1\}^{\leq n}$. We can do so, if and only if $|\mathcal{O}| \leq 2^{n+1} - 1$ as is implied by the following lemma:

Lemma 2.16 For every two non-empty finite sets S, T, there exists a one-to-one $E : S \to T$ if and only if $|S| \leq |T|$.

Proof. Let $k = |S|$ and $m = |T|$ and so write the elements of S and T as $S = \{s_0, s_1, \ldots, s_{k-1}\}$ and $T = \{t_0, t_1, \ldots, t_{m-1}\}$. We need to show that there is a one-to-one function $E : S \to T$ iff $k \leq m$. For the "if" direction, if $k \leq m$ we can simply define $E(s_i) = t_i$ for every $i \in [k]$. Clearly for $i \neq j$, $t_i = E(s_i) \neq E(s_j) = t_j$, and hence this function is one-to-one. In the other direction, suppose that $k > m$ and $E : S \to T$ is some function. Then E cannot be one-to-one. Indeed, for $i = 0, 1, \ldots, m-1$ let us "mark" the element $t_j = E(s_i)$ in T. If t_j was marked before, then we have found two objects in S mapping to the same element t_j. Otherwise, since T has m elements, when we get to $i = m-1$ we mark all the objects in T. Hence, in this case, $E(s_m)$ must map to an element that was already marked before. (This observation is sometimes known as the "Pigeonhole Principle": the principle that

if you have a pigeon coop with m holes and $k > m$ pigeons, then there must be two pigeons in the same hole.)

■

2.5.2 Prefix-free encoding

When showing a representation scheme for rational numbers, we used the "hack" of encoding the alphabet $\{0, 1, \|\}$ to represent tuples of strings as a single string. This is a special case of the general paradigm of *prefix-free* encoding. The idea is the following: if our representation has the property that no string x representing an object o is a *prefix* (i.e., an initial substring) of a string y representing a different object o', then we can represent a *list* of objects by merely concatenating the representations of all the list members. For example, because in English every sentence ends with a punctuation mark such as a period, exclamation, or question mark, no sentence can be a prefix of another and so we can represent a list of sentences by merely concatenating the sentences one after the other. (English has some complications such as periods used for abbreviations (e.g., "e.g.") or sentence quotes containing punctuation, but the high level point of a prefix-free representation for sentences still holds.)

It turns out that we can transform *every* representation to a prefix-free form. This justifies Big Idea 1, and allows us to transform a representation scheme for objects of a type T to a representation scheme of *lists* of objects of the type T. By repeating the same technique, we can also represent lists of lists of objects of type T, and so on and so forth. But first let us formally define prefix-freeness:

Definition 2.17 — Prefix free encoding. For two strings y, y', we say that y is a *prefix* of y' if $|y| \leq |y'|$ and for every $i < |y|$, $y'_i = y_i$.

Let $\mathcal{O}$ be a non-empty set and $E : \mathcal{O} \to \{0,1\}^*$ be a function. We say that E is *prefix-free* if $E(o)$ is non-empty for every $o \in \mathcal{O}$ and there does not exist a distinct pair of objects $o, o' \in \mathcal{O}$ such that $E(o)$ is a prefix of $E(o')$.

Recall that for every set $\mathcal{O}$, the set $\mathcal{O}^*$ consists of all finite length tuples (i.e., *lists*) of elements in $\mathcal{O}$. The following theorem shows that if E is a prefix-free encoding of $\mathcal{O}$ then by concatenating encodings we can obtain a valid (i.e., one-to-one) representation of $\mathcal{O}^*$:

Theorem 2.18 — Prefix-free implies tuple encoding. Suppose that $E : \mathcal{O} \to \{0,1\}^*$ is prefix-free. Then the following map $\overline{E} : \mathcal{O}^* \to \{0,1\}^*$ is one to one, for every $o_0, \ldots, o_{k-1} \in \mathcal{O}^*$, we define

$$\overline{E}(o_0, \ldots, o_{k-1}) = E(o_0)E(o_1)\cdots E(o_{k-1}) \ .$$

P

Theorem 2.18 is an example of a theorem that is a little hard to parse, but in fact is fairly straightforward to prove once you understand what it means. Therefore, I highly recommend that you pause here to make sure you understand the statement of this theorem. You should also try to prove it on your own before proceeding further.

Figure 2.9: If we have a prefix-free representation of each object then we can concatenate the representations of k objects to obtain a representation for the tuple $(o_0, \ldots, o_{k-1})$.

Proof Idea:

The idea behind the proof is simple. Suppose that for example we want to decode a triple (o_0, o_1, o_2) from its representation $x = \overline{E}(o_0, o_1, o_2) = E(o_0)E(o_1)E(o_2)$. We will do so by first finding the first prefix x_0 of x that is a representation of some object. Then we will decode this object, remove x_0 from x to obtain a new string x', and continue onwards to find the first prefix x_1 of x' and so on and so forth (see Exercise 2.9). The prefix-freeness property of E will ensure that x_0 will in fact be $E(o_0)$, x_1 will be $E(o_1)$, etc.

⋆

Proof of Theorem 2.18. We now show the formal proof. Suppose, towards the sake of contradiction, that there exist two distinct tuples $(o_0, \ldots, o_{k-1})$ and $(o'_0, \ldots, o'_{k'-1})$ such that

$$\overline{E}(o_0, \ldots, o_{k-1}) = \overline{E}(o'_0, \ldots, o'_{k'-1}) \; . \tag{2.4}$$

We will denote the string $\overline{E}(o_0, \ldots, o_{k-1})$ by $\overline{x}$.

Let i be the first index such that $o_i \neq o'_i$. (If $o_i = o'_i$ for all i then, since we assume the two tuples are distinct, one of them must be larger than the other. In this case we assume without loss of generality that $k' > k$ and let $i = k$.) In the case that $i < k$, we see that the string $\overline{x}$ can be written in two different ways:

$$\overline{x} = \overline{E}(o_0, \ldots, o_{k-1}) = x_0 \cdots x_{i-1} E(o_i) E(o_{i+1}) \cdots E(o_{k-1})$$

and

$$\overline{x} = \overline{E}(o'_0, \ldots, o'_{k'-1}) = x_0 \cdots x_{i-1} E(o'_i) E(o'_{i+1}) \cdots E(o'_{k'-1})$$

where $x_j = E(o_j) = E(o'_j)$ for all $j < i$. Let $\overline{y}$ be the string obtained after removing the prefix $x_0 \cdots x_{i-1}$ from $\overline{x}$. We see that $\overline{y}$ can be written as both $\overline{y} = E(o_i)s$ for some string $s \in \{0,1\}^*$ and as $\overline{y} = E(o'_i)s'$ for some $s' \in \{0,1\}^*$. But this means that one of $E(o_i)$ and $E(o'_i)$ must be a prefix of the other, contradicting the prefix-freeness of E.

In the case that $i = k$ and $k' > k$, we get a contradiction in the following way. In this case

$$\overline{x} = E(o_0)\cdots E(o_{k-1}) = E(o_0)\cdots E(o_{k-1})E(o'_k)\cdots E(o'_{k'-1})$$

which means that $E(o'_k)\cdots E(o'_{k'-1})$ must correspond to the empty string "". But in such a case $E(o'_k)$ must be the empty string, which in particular is the prefix of any other string, contradicting the prefix-freeness of E.

■

R

Remark 2.19 — Prefix freeness of list representation. Even if the representation E of objects in $\mathcal{O}$ is prefix free, this does not mean that our representation $\overline{E}$ of *lists* of such objects will be prefix free as well. In fact, it won't be: for every three objects o, o', o'' the representation of the list (o, o') will be a prefix of the representation of the list (o, o', o''). However, as we see in Lemma 2.20 below, we can transform *every* representation into prefix-free form, and so will be able to use that transformation if needed to represent lists of lists, lists of lists of lists, and so on and so forth.

2.5.3 Making representations prefix-free

Some natural representations are prefix-free. For example, every *fixed output length* representation (i.e., one-to-one function $E : \mathcal{O} \to \{0,1\}^n$) is automatically prefix-free, since a string x can only be a prefix of an equal-length x' if x and x' are identical. Moreover, the approach we used for representing rational numbers can be used to show the following:

Lemma 2.20 Let $E : \mathcal{O} \to \{0,1\}^*$ be a one-to-one function. Then there is a one-to-one prefix-free encoding $\overline{E}$ such that $|\overline{E}(o)| \leq 2|E(o)| + 2$ for every $o \in \mathcal{O}$.

P

For the sake of completeness, we will include the proof below, but it is a good idea for you to pause here and try to prove it on your own, using the same technique we used for representing rational numbers.

Proof of Lemma 2.20. The idea behind the proof is to use the map $0 \mapsto 00$, $1 \mapsto 11$ to "double" every bit in the string x and then mark the end of the string by concatenating to it the pair 01. If we encode a string x in this way, it ensures that the encoding of x is never a prefix

of the encoding of a distinct string x'. Formally, we define the function $PF : \{0,1\}^* \to \{0,1\}^*$ as follows:

$$PF(x) = x_0 x_0 x_1 x_1 \ldots x_{n-1} x_{n-1} 01$$

for every $x \in \{0,1\}^*$. If $E : \mathcal{O} \to \{0,1\}^*$ is the (potentially not prefix-free) representation for $\mathcal{O}$, then we transform it into a prefix-free representation $\overline{E} : \mathcal{O} \to \{0,1\}^*$ by defining $\overline{E}(o) = PF(E(o))$.

To prove the lemma we need to show that **(1)** $\overline{E}$ is one-to-one and **(2)** $\overline{E}$ is prefix-free. In fact, prefix freeness is a stronger condition than one-to-one (if two strings are equal then in particular one of them is a prefix of the other) and hence it suffices to prove **(2)**, which we now do.

Let $o \neq o'$ in $\mathcal{O}$ be two distinct objects. We will prove that $\overline{E}(o)$ is not a prefix of $\overline{E}(o')$, or in other words $PF(x)$ is not a prefix of $PF(x')$ where $x = E(o)$ and $x' = E(o')$. Since E is one-to-one, $x \neq x'$. We will split into three cases, depending on whether $|x| < |x'|$, $|x| = |x'|$, or $|x| > |x'|$. If $|x| < |x'|$ then the two bits in positions $2|x|, 2|x|+1$ in $PF(x)$ have the value 01 but the corresponding bits in $PF(x')$ will equal either 00 or 11 (depending on the $|x|$-th bit of x') and hence $PF(x)$ cannot be a prefix of $PF(x')$. If $|x| = |x'|$ then, since $x \neq x'$, there must be a coordinate i in which they differ, meaning that the strings $PF(x)$ and $PF(x')$ differ in the coordinates $2i, 2i+1$, which again means that $PF(x)$ cannot be a prefix of $PF(x')$. If $|x| > |x'|$ then $|PF(x)| = 2|x| + 2 > |PF(x')| = 2|x'| + 2$ and hence $PF(x)$ is longer than (and cannot be a prefix of) $PF(x')$. In all cases we see that $PF(x) = \overline{E}(o)$ is not a prefix of $PF(x') = \overline{E}(o')$, hence completing the proof.

■

The proof of Lemma 2.20 is not the only or even the best way to transform an arbitrary representation into prefix-free form. Exercise 2.10 asks you to construct a more efficient prefix-free transformation satisfying $|\overline{E}(o)| \leq |E(o)| + O(\log |E(o)|)$.

2.5.4 "Proof by Python" (optional)

The proofs of Theorem 2.18 and Lemma 2.20 are *constructive* in the sense that they give us:

- A way to transform the encoding and decoding functions of any representation of an object O to encoding and decoding functions that are prefix-free, and
- A way to extend prefix-free encoding and decoding of single objects to encoding and decoding of *lists* of objects by concatenation.

Specifically, we could transform any pair of Python functions `encode` and `decode` to functions `pfencode` and `pfdecode` that correspond to a prefix-free encoding and decoding. Similarly, given `pfencode` and `pfdecode` for single objects, we can extend them to encoding of lists. Let us show how this works for the case of the `NtS` and `StN` functions we defined above.

We start with the "Python proof" of Lemma 2.20: a way to transform an arbitrary representation into one that is *prefix free*. The function `prefixfree` below takes as input a pair of encoding and decoding functions, and returns a triple of functions containing *prefix-free* encoding and decoding functions, as well as a function that checks whether a string is a valid encoding of an object.

```
# takes functions encode and decode mapping
# objects to lists of bits and vice versa,
# and returns functions pfencode and pfdecode that
# maps objects to lists of bits and vice versa
# in a prefix-free way.
# Also returns a function pfvalid that says
# whether a list is a valid encoding
def prefixfree(encode, decode):
    def pfencode(o):
        L = encode(o)
        return [L[i//2] for i in range(2*len(L))]+[0,1]
    def pfdecode(L):
        return decode([L[j] for j in range(0,len(L)-2,2)])
    def pfvalid(L):
        return (len(L) % 2 == 0 ) and all(L[2*i]==L[2*i+1]
         ↪  for i in range((len(L)-2)//2)) and
         ↪  L[-2:]==[0,1]

    return pfencode, pfdecode, pfvalid

pfNtS, pfStN , pfvalidN = prefixfree(NtS,StN)

NtS(234)
# 11101010
pfNtS(234)
# 111111001100110001
pfStN(pfNtS(234))
# 234
pfvalidM(pfNtS(234))
# true
```

Note that the Python function `prefixfree` above takes two *Python functions* as input and outputs three Python functions as output. (When it's not too awkward, we use the term "Python function" or "subroutine" to distinguish between such snippets of Python programs and mathematical functions.) You don't have to know Python in this course, but you do need to get comfortable with the idea of functions as mathematical objects in their own right, that can be used as inputs and outputs of other functions.

We now show a "Python proof" of Theorem 2.18. Namely, we show a function `represlists` that takes as input a prefix-free representation scheme (implemented via encoding, decoding, and validity testing functions) and outputs a representation scheme for *lists* of such objects. If we want to make this representation prefix-free then we could fit it into the function `prefixfree` above.

```
def represlists(pfencode,pfdecode,pfvalid):
    """
    Takes functions pfencode, pfdecode and pfvalid,
    and returns functions encodelists, decodelists
    that can encode and decode lists of the objects
    respectively.
    """

    def encodelist(L):
        """Gets list of objects, encodes it as list of
        ↪ bits"""
        return "".join([pfencode(obj) for obj in L])

    def decodelist(S):
        """Gets lists of bits, returns lists of objects"""
        i=0; j=1 ; res = []
        while j<=len(S):
            if pfvalid(S[i:j]):
                res += [pfdecode(S[i:j])]
                i=j
            j+= 1
        return res

    return encodelist,decodelist


LtS , StL = represlists(pfNtS,pfStN,pfvalidN)
```

```
LtS([234,12,5])
# 11111100110011000111110000011100l101
StL(LtS([234,12,5]))
# [234, 12, 5]
```

2.5.5 Representing letters and text

We can represent a letter or symbol by a string, and then if this representation is prefix-free, we can represent a sequence of symbols by merely concatenating the representation of each symbol. One such representation is the ASCII that represents 128 letters and symbols as strings of 7 bits. Since the ASCII representation is fixed-length, it is automatically prefix-free (can you see why?). Unicode is the representation of (at the time of this writing) about 128,000 symbols as numbers (known as *code points*) between 0 and $1,114,111$. There are several types of prefix-free representations of the code points, a popular one being UTF-8 that encodes every codepoint into a string of length between 8 and 32.

■ **Example 2.21 — The Braille representation.** The *Braille system* is another way to encode letters and other symbols as binary strings. Specifically, in Braille, every letter is encoded as a string in $\{0,1\}^6$, which is written using indented dots arranged in two columns and three rows, see Fig. 2.10. (Some symbols require more than one six-bit string to encode, and so Braille uses a more general prefix-free encoding.)

The Braille system was invented in 1821 by Louis Braille when he was just 12 years old (though he continued working on it and improving it throughout his life). Braille was a French boy who lost his eyesight at the age of 5 as the result of an accident.

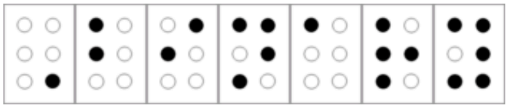


Figure 2.10: The word "Binary" in "Grade 1" or "uncontracted" Unified English Braille. This word is encoded using seven symbols since the first one is a modifier indicating that the first letter is capitalized.

■ **Example 2.22 — Representing objects in C (optional).** We can use programming languages to probe how our computing environment represents various values. This is easiest to do in "unsafe" programming languages such as `C` that allow direct access to the memory.

Using a simple `C` program we have produced the following representations of various values. One can see that for integers, multiplying by 2 corresponds to a "left shift" inside each byte. In contrast, for floating-point numbers, multiplying by two corresponds to adding one to the exponent part of the representation. In the architecture we used, a negative number is represented

using the two's complement approach. `C` represents strings in a prefix-free form by ensuring that a zero byte is at their end.

```
int      2    : 00000010 00000000 00000000 00000000
int      4    : 00000100 00000000 00000000 00000000
int      513  : 00000001 00000010 00000000 00000000
long     513  : 00000001 00000010 00000000 00000000
 ↪  00000000 00000000 00000000 00000000
int      -1   : 11111111 11111111 11111111 11111111
int      -2   : 11111110 11111111 11111111 11111111
string   Hello: 01001000 01100101 01101100 01101100
 ↪  01101111 00000000
string   abcd : 01100001 01100010 01100011 01100100
 ↪  00000000
float    33.0 : 00000000 00000000 00000100 01000010
float    66.0 : 00000000 00000000 10000100 01000010
float    132.0: 00000000 00000000 00000100 01000011
double   132.0: 00000000 00000000 00000000 00000000
 ↪  00000000 10000000 01100000 01000000
```

2.5.6 Representing vectors, matrices, images

Once we can represent numbers and lists of numbers, then we can also represent *vectors* (which are just lists of numbers). Similarly, we can represent lists of lists, and thus, in particular, can represent *matrices*. To represent an image, we can represent the color at each pixel by a list of three numbers corresponding to the intensity of Red, Green and Blue. (We can restrict to three primary colors since most humans only have three types of cones in their retinas; we would have needed 16 primary colors to represent colors visible to the Mantis Shrimp.) Thus an image of n pixels would be represented by a list of n such length-three lists. A video can be represented as a list of images. Of course these representations are rather wasteful and much more compact representations are typically used for images and videos, though this will not be our concern in this book.

2.5.7 Representing graphs

A *graph* on n vertices can be represented as an $n \times n$ *adjacency* matrix whose $(i,j)^{th}$ entry is equal to 1 if the edge (i,j) is present and is equal to 0 otherwise. That is, we can represent an n vertex directed graph $G = (V,E)$ as a string $A \in \{0,1\}^{n^2}$ such that $A_{i,j} = 1$ iff the edge $\overrightarrow{i\ j} \in E$. We can transform an undirected graph to a directed graph by replacing every edge $\{i,j\}$ with both edges $\overrightarrow{i\ j}$ and $\overleftarrow{i\ j}$

Another representation for graphs is the *adjacency list* representation. That is, we identify the vertex set V of a graph with the set $[n]$

where $n = |V|$, and represent the graph $G = (V, E)$ as a list of n lists, where the i-th list consists of the out-neighbors of vertex i. The difference between these representations can be significant for some applications, though for us would typically be immaterial.

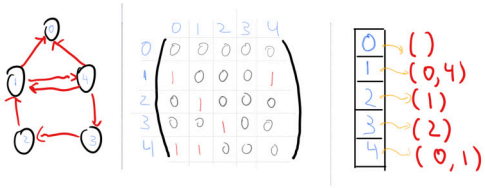


Figure 2.11: Representing the graph $G = (\{0,1,2,3,4\}, \{(1,0),(4,0),(1,4),(4,1),(2,1),(3,2),(4,3)\})$ in the adjacency matrix and adjacency list representations.

2.5.8 Representing lists and nested lists

If we have a way of representing objects from a set $\mathcal{O}$ as binary strings, then we can represent lists of these objects by applying a prefix-free transformation. Moreover, we can use a trick similar to the above to handle *nested* lists. The idea is that if we have some representation $E : \mathcal{O} \to \{0,1\}^*$, then we can represent nested lists of items from $\mathcal{O}$ using strings over the five element alphabet $\Sigma = \{$ `0`,`1`,`[`,`]`,`,` $\}$. For example, if o_1 is represented by `0011`, o_2 is represented by `10011`, and o_3 is represented by `00111`, then we can represent the nested list $(o_1, (o_2, o_3))$ as the string `"[0011,[10011,00111]]"` over the alphabet Σ. By encoding every element of Σ itself as a three-bit string, we can transform any representation for objects $\mathcal{O}$ into a representation that enables representing (potentially nested) lists of these objects.

2.5.9 Notation

We will typically identify an object with its representation as a string. For example, if $F : \{0,1\}^* \to \{0,1\}^*$ is some function that maps strings to strings and n is an integer, we might make statements such as "$F(n) + 1$ is prime" to mean that if we represent n as a string x, then the integer m represented by the string $F(x)$ satisfies that $m + 1$ is prime. (You can see how this convention of identifying objects with their representation can save us a lot of cumbersome formalism.) Similarly, if x, y are some objects and F is a function that takes strings as inputs, then by $F(x, y)$ we will mean the result of applying F to the representation of the ordered pair (x, y). We use the same notation to invoke functions on k-tuples of objects for every k.

This convention of identifying an object with its representation as a string is one that we humans follow all the time. For example, when people say a statement such as "17 is a prime number", what they really mean is that the integer whose decimal representation is the string "17", is prime.

When we say

A is an algorithm that computes the multiplication function on natural numbers.

what we really mean is that

A is an algorithm that computes the function $F : \{0,1\}^* \to \{0,1\}^*$ such that for every pair $a, b \in \mathbb{N}$, if $x \in \{0,1\}^*$ is a string representing the pair (a, b) then $F(x)$ will be a string representing their product $a \cdot b$.

2.6 DEFINING COMPUTATIONAL TASKS AS MATHEMATICAL FUNCTIONS

Abstractly, a *computational process* is some process that takes an input which is a string of bits and produces an output which is a string of bits. This transformation of input to output can be done using a modern computer, a person following instructions, the evolution of some natural system, or any other means.

In future chapters, we will turn to mathematically defining computational processes, but, as we discussed above, at the moment we focus on *computational tasks*. That is, we focus on the **specification** and not the **implementation**. Again, at an abstract level, a computational task can specify any relation that the output needs to have with the input. However, for most of this book, we will focus on the simplest and most common task of *computing a function*. Here are some examples:

- Given (a representation of) two integers x, y, compute the product $x \times y$. Using our representation above, this corresponds to computing a function from $\{0,1\}^*$ to $\{0,1\}^*$. We have seen that there is more than one way to solve this computational task, and in fact, we still do not know the best algorithm for this problem.

- Given (a representation of) an integer $z > 1$, compute its *factorization*; i.e., the list of primes $p_1 \leq \cdots \leq p_k$ such that $z = p_1 \cdots p_k$. This again corresponds to computing a function from $\{0,1\}^*$ to $\{0,1\}^*$. The gaps in our knowledge of the complexity of this problem are even larger.

- Given (a representation of) a graph G and two vertices s and t, compute the length of the shortest path in G between s and t, or do the same for the *longest* path (with no repeated vertices) between s and t. Both these tasks correspond to computing a function from $\{0,1\}^*$ to $\{0,1\}^*$, though it turns out that there is a vast difference in their computational difficulty.

- Given the code of a Python program, determine whether there is an input that would force it into an infinite loop. This task corresponds to computing a *partial* function from $\{0,1\}^*$ to $\{0,1\}$ since not every string corresponds to a syntactically valid Python program. We will see that we *do* understand the computational status of this problem, but the answer is quite surprising.

- Given (a representation of) an image I, decide if I is a photo of a cat or a dog. This corresponds to computing some (partial) function from $\{0,1\}^*$ to $\{0,1\}$.

R

Remark 2.23 — Boolean functions and languages. An important special case of computational tasks corresponds to computing *Boolean* functions, whose output is a single bit $\{0,1\}$. Computing such functions corresponds to answering a YES/NO question, and hence this task is also known as a *decision problem*. Given any function $F : \{0,1\}^* \to \{0,1\}$ and $x \in \{0,1\}^*$, the task of computing $F(x)$ corresponds to the task of deciding whether or not $x \in L$ where $L = \{x : F(x) = 1\}$ is known as the *language* that corresponds to the function F. (The language terminology is due to historical connections between the theory of computation and formal linguistics as developed by Noam Chomsky.) Hence many texts refer to such a computational task as *deciding a language*.

For every particular function F, there can be several possible *algorithms* to compute F. We will be interested in questions such as:

- For a given function F, can it be the case that *there is no algorithm* to compute F?
- If there is an algorithm, what is the best one? Could it be that F is "effectively uncomputable" in the sense that every algorithm for computing F requires a prohibitively large amount of resources?
- If we cannot answer this question, can we show equivalence between different functions F and F' in the sense that either they are both easy (i.e., have fast algorithms) or they are both hard?
- Can a function being hard to compute ever be a *good thing*? Can we use it for applications in areas such as cryptography?

In order to do that, we will need to mathematically define the notion of an *algorithm*, which is what we will do in Chapter 3.

Figure 2.12: A subset $L \subseteq \{0,1\}^*$ can be identified with the function $F : \{0,1\}^* \to \{0,1\}$ such that $F(x) = 1$ if $x \in L$ and $F(x) = 0$ if $x \notin L$. Functions with a single bit of output are called *Boolean functions*, while subsets of strings are called *languages*. The above shows that the two are essentially the same object, and we can identify the task of deciding membership in L (known as *deciding a language* in the literature) with the task of computing the function F.

2.6.1 Distinguish functions from programs!

You should always watch out for potential confusions between **specifications** and **implementations** or equivalently between **mathematical functions** and **algorithms/programs**. It does not help that programming languages (Python included) use the term *"functions"* to denote (parts of) *programs*. This confusion also stems from thousands of years of mathematical history, where people typically defined functions by means of a way to compute them.

For example, consider the multiplication function on natural numbers. This is the function $MULT : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ that maps a pair (x, y) of natural numbers to the number $x \cdot y$. As we mentioned, it can be implemented in more than one way:

```python
def mult1(x,y):
    res = 0
    while y>0:
        res += x
        y   -= 1
    return res

def mult2(x,y):
    a = str(x) # represent x as string in decimal notation
    b = str(y) # represent y as string in decimal notation
    res = 0
    for i in range(len(a)):
        for j in range(len(b)):
            res += int(a[len(a)-i])*int(b[len(b)-
             ↪  j])*(10**(i+j))
    return res

print(mult1(12,7))
# 84
print(mult2(12,7))
# 84
```

Both `mult1` and `mult2` produce the same output given the same pair of natural number inputs. (Though `mult1` will take far longer to do so when the numbers become large.) Hence, even though these are two different *programs*, they compute the same *mathematical function*. This distinction between a *program* or *algorithm* A, and the *function* F that A *computes* will be absolutely crucial for us in this course (see also Fig. 2.13).

Big Idea 2 A *function* is not the same as a *program*. A program *computes* a function.

Distinguishing *functions* from *programs* (or other ways for computing, including *circuits* and *machines*) is a crucial theme for this course. For this reason, this is often a running theme in questions that I (and many other instructors) assign in homework and exams (hint, hint).

Remark 2.24 — Computation beyond functions (advanced, optional). Functions capture quite a lot of computational tasks, but one can consider more general settings as well. For starters, we can and will talk about *partial* functions, that are not defined on all inputs. When computing a partial function, we only

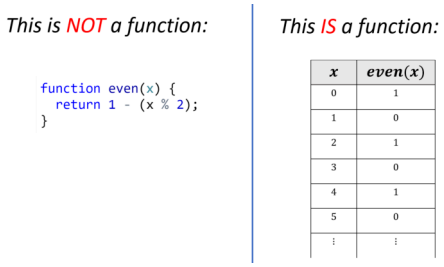


Figure 2.13: A *function* is a mapping of inputs to outputs. A *program* is a set of instructions on how to obtain an output given an input. A program *computes* a function, but it is not the same as a function, popular programming language terminology notwithstanding.

need to worry about the inputs on which the function is defined. Another way to say it is that we can design an algorithm for a partial function F under the assumption that someone "promised" us that all inputs x would be such that $F(x)$ is defined (as otherwise, we do not care about the result). Hence such tasks are also known as *promise problems*.

Another generalization is to consider *relations* that may have more than one possible admissible output. For example, consider the task of finding any solution for a given set of equations. A *relation* R maps a string $x \in \{0,1\}^*$ into a *set of strings* $R(x)$ (for example, x might describe a set of equations, in which case $R(x)$ would correspond to the set of all solutions to x). We can also identify a relation R with the set of pairs of strings (x, y) where $y \in R(x)$. A computational process solves a relation if for every $x \in \{0,1\}^*$, it outputs some string $y \in R(x)$.

Later in this book, we will consider even more general tasks, including *interactive* tasks, such as finding a good strategy in a game, tasks defined using probabilistic notions, and others. However, for much of this book, we will focus on the task of computing a function, and often even a *Boolean* function, that has only a single bit of output. It turns out that a great deal of the theory of computation can be studied in the context of this task, and the insights learned are applicable in the more general settings.

Chapter Recap

- We can represent objects we want to compute on using binary strings.
- A representation scheme for a set of objects $\mathcal{O}$ is a one-to-one map from $\mathcal{O}$ to $\{0,1\}^*$.
- We can use prefix-free encoding to "boost" a representation for a set $\mathcal{O}$ into representations of lists of elements in $\mathcal{O}$.
- A basic computational task is the task of *computing a function* $F : \{0,1\}^* \to \{0,1\}^*$. This task encompasses not just arithmetical computations such as multiplication, factoring, etc. but a great many other tasks arising in areas as diverse as scientific computing, artificial intelligence, image processing, data mining and many many more.
- We will study the question of finding (or at least giving bounds on) what the *best* algorithm for computing F for various interesting functions F is.

2.7 EXERCISES

Exercise 2.1 Which one of these objects can be represented by a binary string?

a. An integer x

b. An undirected graph G.

c. A directed graph H

d. All of the above.

■

Exercise 2.2 — Binary representation. a. Prove that the function $NtS : \mathbb{N} \to \{0,1\}^*$ of the binary representation defined in (2.1) satisfies that for every $n \in \mathbb{N}$, if $x = NtS(n)$ then $|x| = 1 + \max(0, \lfloor \log_2 n \rfloor)$ and $x_i = \lfloor x/2^{\lfloor \log_2 n \rfloor - i} \rfloor \mod 2$.

b. Prove that NtS is a one to one function by coming up with a function $StN : \{0,1\}^* \to \mathbb{N}$ such that $StN(NtS(n)) = n$ for every $n \in \mathbb{N}$.

■

Exercise 2.3 — More compact than ASCII representation. The ASCII encoding can be used to encode a string of n English letters as a $7n$ bit binary string, but in this exercise, we ask about finding a more compact representation for strings of English lowercase letters.

1. Prove that there exists a representation scheme (E, D) for strings over the 26-letter alphabet $\{a, b, c, \ldots, z\}$ as binary strings such that for every $n > 0$ and length-n string $x \in \{a, b, \ldots, z\}^n$, the representation $E(x)$ is a binary string of length at most $4.8n + 1000$. In other words, prove that for every n, there exists a one-to-one function $E : \{a, b, \ldots, z\}^n \to \{0,1\}^{\lfloor 4.8n+1000 \rfloor}$.

2. Prove that there exists *no* representation scheme for strings over the alphabet $\{a, b, \ldots, z\}$ as binary strings such that for every length-n string $x \in \{a, b, \ldots, z\}^n$, the representation $E(x)$ is a binary string of length $\lfloor 4.6n + 1000 \rfloor$. In other words, prove that there exists some $n > 0$ such that there is no one-to-one function $E : \{a, b, \ldots, z\}^n \to \{0,1\}^{\lfloor 4.6n+1000 \rfloor}$.

3. Python's `bz2.compress` function is a mapping from strings to strings, which uses the *lossless* (and hence *one to one*) bzip2 algorithm for compression. After converting to lowercase, and truncating spaces and numbers, the text of Tolstoy's "War and Peace" contains $n = 2,517,262$. Yet, if we run `bz2.compress` on the string of the text of "War and Peace" we get a string of length $k = 6,274,768$

bits, which is only $2.49n$ (and in particular much smaller than $4.6n$). Explain why this does not contradict your answer to the previous question.

4. Interestingly, if we try to apply `bz2.compress` on a *random* string, we get much worse performance. In my experiments, I got a ratio of about 4.78 between the number of bits in the output and the number of characters in the input. However, one could imagine that one could do better and that there exists a company called "Pied Piper" with an algorithm that can losslessly compress a string of n random lowercase letters to fewer than $4.6n$ bits.[5] Show that this is not the case by proving that for *every* $n > 100$ and one to one function $Encode : \{a, \ldots, z\}^n \rightarrow \{0,1\}^*$, if we let Z be the random variable $|Encode(x)|$ (i.e., the length of $Encode(x)$) for x chosen uniformly at random from the set $\{a, \ldots, z\}^n$, then the expected value of Z is at least $4.6n$.

[5] Actually that particular fictional company uses a metric that focuses more on compression *speed* then *ratio*, see here and here.

■

Exercise 2.4 — Representing graphs: upper bound. Show that there is a string representation of directed graphs with vertex set $[n]$ and degree at most 10 that uses at most $1000n \log n$ bits. More formally, show the following: Suppose we define for every $n \in \mathbb{N}$, the set G_n as the set containing all directed graphs (with no self loops) over the vertex set $[n]$ where every vertex has degree at most 10. Then, prove that for every sufficiently large n, there exists a one-to-one function $E : G_n \rightarrow \{0,1\}^{\lfloor 1000n \log n \rfloor}$.

■

Exercise 2.5 — Representing graphs: lower bound. 1. Define S_n to be the set of one-to-one and onto functions mapping $[n]$ to $[n]$. Prove that there is a one-to-one mapping from S_n to G_{2n}, where G_{2n} is the set defined in Exercise 2.4 above.

2. In this question you will show that one cannot improve the representation of Exercise 2.4 to length $o(n \log n)$. Specifically, prove for every sufficiently large $n \in \mathbb{N}$ there is *no* one-to-one function $E : G_n \rightarrow \{0,1\}^{\lfloor 0.001n \log n \rfloor + 1000}$.

■

Exercise 2.6 — Multiplying in different representation. Recall that the grade-school algorithm for multiplying two numbers requires $O(n^2)$ operations. Suppose that instead of using decimal representation, we use one of the following representations $R(x)$ to represent a number x between 0 and $10^n - 1$. For which one of these representations you can still multiply the numbers in $O(n^2)$ operations?

a. The standard binary representation: $B(x) = (x_0, \ldots, x_k)$ where $x = \sum_{i=0}^{k} x_i 2^i$ and k is the largest number s.t. $x \geq 2^k$.

b. The reverse binary representation: $B(x) = (x_k, \ldots, x_0)$ where x_i is defined as above for $i = 0, \ldots, k-1$.

c. Binary coded decimal representation: $B(x) = (y_0, \ldots, y_{n-1})$ where $y_i \in \{0,1\}^4$ represents the ith decimal digit of x mapping 0 to 0000, 1 to 0001, 2 to 0010, etc. (i.e. 9 maps to 1001)

d. All of the above.

■

Exercise 2.7 Suppose that $R : \mathbb{N} \to \{0,1\}^*$ corresponds to representing a number x as a string of x 1's, (e.g., $R(4) = 1111$, $R(7) = 1111111$, etc.). If x, y are numbers between 0 and $10^n - 1$, can we still multiply x and y using $O(n^2)$ operations if we are given them in the representation $R(\cdot)$?

■

Exercise 2.8 Recall that if F is a one-to-one and onto function mapping elements of a finite set U into a finite set V then the sizes of U and V are the same. Let $B : \mathbb{N} \to \{0,1\}^*$ be the function such that for every $x \in \mathbb{N}$, $B(x)$ is the binary representation of x.

1. Prove that $x < 2^k$ if and only if $|B(x)| \leq k$.

2. Use 1. to compute the size of the set $\{y \in \{0,1\}^* : |y| \leq k\}$ where $|y|$ denotes the length of the string y.

3. Use 1. and 2. to prove that $2^k - 1 = 1 + 2 + 4 + \cdots + 2^{k-1}$.

■

Exercise 2.9 — Prefix-free encoding of tuples. Suppose that $F : \mathbb{N} \to \{0,1\}^*$ is a one-to-one function that is *prefix-free* in the sense that there is no $a \neq b$ s.t. $F(a)$ is a prefix of $F(b)$.

a. Prove that $F_2 : \mathbb{N} \times \mathbb{N} \to \{0,1\}^*$, defined as $F_2(a,b) = F(a)F(b)$ (i.e., the concatenation of $F(a)$ and $F(b)$) is a one-to-one function.

b. Prove that $F_* : \mathbb{N}^* \to \{0,1\}^*$ defined as $F_*(a_1, \ldots, a_k) = F(a_1) \cdots F(a_k)$ is a one-to-one function, where $\mathbb{N}^*$ denotes the set of all finite-length lists of natural numbers.

■

Exercise 2.10 — More efficient prefix-free transformation. Suppose that $F : O \to \{0,1\}^*$ is some (not necessarily prefix-free) representation of the objects in the set O, and $G : \mathbb{N} \to \{0,1\}^*$ is a prefix-free representation of the natural numbers. Define $F'(o) = G(|F(o)|)F(o)$ (i.e., the concatenation of the representation of the length $F(o)$ and $F(o)$).

a. Prove that F' is a prefix-free representation of O.

b. Show that we can transform any representation to a prefix-free one by a modification that takes a k bit string into a string of length at most $k + O(\log k)$.

c. Show that we can transform any representation to a prefix-free one by a modification that takes a k bit string into a string of length at most $k + \log k + O(\log \log k)$.[6]

[6] Hint: Think recursively how to represent the length of the string.

■

Exercise 2.11 — Kraft's Inequality. Suppose that $S \subseteq \{0,1\}^*$ is some finite prefix-free set, and let n some number larger than $\max\{|x| : x \in X\}$.

a. For every $x \in S$, let $L(x) \subseteq \{0,1\}^n$ denote all the length-n strings whose first k bits are $x_0, \ldots, x_{k-1}$. Prove that **(1)** $|L(x)| = 2^{n-|x|}$ and **(2)** For every distinct $x, x' \in S$, $L(x)$ is disjoint from $L(x')$.

b. Prove that $\sum_{x \in S} 2^{-|x|} \leq 1$. (*Hint:* first show that $\sum_{x \in S} |L(x)| \leq 2^n$.)

c. Prove that there is no prefix-free encoding of strings with less than logarithmic overhead. That is, prove that there is no function $PF : \{0,1\}^* \to \{0,1\}^*$ s.t. $|PF(x)| \leq |x| + 0.9 \log |x|$ for every sufficiently large $x \in \{0,1\}^*$ and such that the set $\{PF(x) : x \in \{0,1\}^*\}$ is prefix-free. The factor 0.9 is arbitrary; all that matters is that it is less than 1.

■

Exercise 2.12 — Composition of one-to-one functions. Prove that for every two one-to-one functions $F : S \to T$ and $G : T \to U$, the function $H : S \to U$ defined as $H(x) = G(F(x))$ is one to one.

■

Exercise 2.13 — Natural numbers and strings. 1. We have shown that the natural numbers can be represented as strings. Prove that the other direction holds as well: that there is a one-to-one map $StN : \{0,1\}^* \to \mathbb{N}$. ($StN$ stands for "strings to numbers.")

2. Recall that Cantor proved that there is no one-to-one map $RtN : \mathbb{R} \to \mathbb{N}$. Show that Cantor's result implies Theorem 2.5.

■

Exercise 2.14 — Map lists of integers to a number. Recall that for every set S, the set S^* is defined as the set of all finite sequences of members of S (i.e., $S^* = \{(x_0, \ldots, x_{n-1}) \mid n \in \mathbb{N} \ , \ \forall_{i \in [n]} x_i \in S\}$). Prove that there is a one-one-map from $\mathbb{Z}^*$ to $\mathbb{N}$ where $\mathbb{Z}$ is the set of $\{\ldots, -3, -2, -1, 0, +1, +2, +3, \ldots\}$ of all integers.

■

2.8 BIBLIOGRAPHICAL NOTES

The study of representing data as strings, including issues such as *compression* and *error corrections* falls under the purview of *information theory*, as covered in the classic textbook of Cover and Thomas [CT06]. Representations are also studied in the field of *data structures design*, as covered in texts such as [Cor+09].

The question of whether to represent integers with the most significant digit first or last is known as Big Endian vs. Little Endian representation. This terminology comes from Cohen's [Coh81] entertaining and informative paper about the conflict between adherents of both schools which he compared to the warring tribes in Jonathan Swift's *"Gulliver's Travels"*. The two's complement representation of signed integers was suggested in von Neumann's classic report [Neu45] that detailed the design approaches for a stored-program computer, though similar representations have been used even earlier in abacus and other mechanical computation devices.

The idea that we should separate the *definition* or *specification* of a function from its *implementation* or *computation* might seem "obvious," but it took quite a lot of time for mathematicians to arrive at this viewpoint. Historically, a function F was identified by rules or formulas showing how to derive the output from the input. As we discuss in greater depth in Chapter 9, in the 1800s this somewhat informal notion of a function started "breaking at the seams," and eventually mathematicians arrived at the more rigorous definition of a function as an arbitrary assignment of input to outputs. While many functions may be described (or computed) by one or more formulas, today we do not consider that to be an essential property of functions, and also allow functions that do not correspond to any "nice" formula.

We have mentioned that all representations of the real numbers are inherently *approximate*. Thus an important endeavor is to understand what guarantees we can offer on the approximation quality of the output of an algorithm, as a function of the approximation quality of the inputs. This question is known as the question of determining the numerical stability of given equations. The Floating-Point Guide website contains an extensive description of the floating-point representation, as well the many ways in which it could subtly fail, see also the website 0.30000000000000004.com.

Dauben [Dau90] gives a biography of Cantor with emphasis on the development of his mathematical ideas. [Hal60] is a classic textbook on set theory, also including Cantor's theorem. Cantor's Theorem is also covered in many texts on discrete mathematics, including [LLM18; LZ19].

The adjacency matrix representation of graphs is not merely a convenient way to map a graph into a binary string, but it turns out that many natural notions and operations on matrices are useful for graphs as well. (For example, Google's PageRank algorithm relies on this viewpoint.) The notes of Spielman's course are an excellent source for this area, known as *spectral graph theory*. We will return to this view much later in this book when we talk about *random walks*.

I

FINITE COMPUTATION

Learning Objectives:

- See that computation can be precisely modeled.
- Learn the computational model of *Boolean circuits / straight-line programs*.
- Equivalence of circuits and straight-line programs.
- Equivalence of AND/OR/NOT and NAND.
- Examples of computing in the physical world.

3
Defining computation

"there is no reason why mental as well as bodily labor should not be economized by the aid of machinery", Charles Babbage, 1852

"If, unwarned by my example, any man shall undertake and shall succeed in constructing an engine embodying in itself the whole of the executive department of mathematical analysis upon different principles or by simpler mechanical means, I have no fear of leaving my reputation in his charge, for he alone will be fully able to appreciate the nature of my efforts and the value of their results.", Charles Babbage, 1864

"To understand a program you must become both the machine and the program.", Alan Perlis, 1982

People have been computing for thousands of years, with aids that include not just pen and paper, but also abacus, slide rules, various mechanical devices, and modern electronic computers. A priori, the notion of computation seems to be tied to the particular mechanism that you use. You might think that the "best" algorithm for multiplying numbers will differ if you implement it in *Python* on a modern laptop than if you use pen and paper. However, as we saw in the introduction (Chapter 0), an algorithm that is asymptotically better would eventually beat a worse one regardless of the underlying technology. This gives us hope for a *technology independent* way of defining computation. This is what we do in this chapter. We will define the notion of computing an output from an input by applying a sequence of basic operations (see Fig. 3.3). Using this, we will be able to precisely define statements such as "function f can be computed by model X" or "function f can be computed by model X using s operations".

This chapter: A non-mathy overview

The main takeaways from this chapter are:

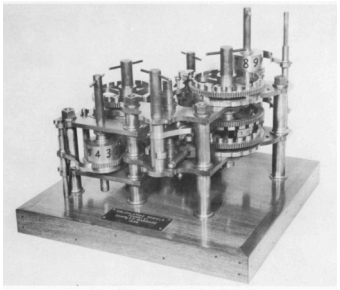


Figure 3.1: Calculating wheels by Charles Babbage. Image taken from the Mark I 'operating manual'

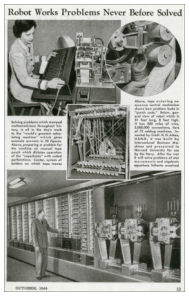


Figure 3.2: A 1944 *Popular Mechanics* article on the Harvard Mark I computer.

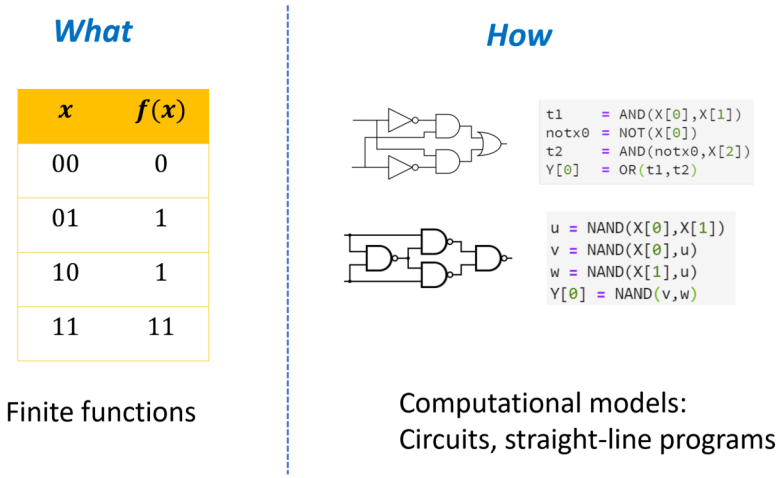


Figure 3.3: A function mapping strings to strings *specifies* a computational task, i.e., describes *what* the desired relation between the input and the output is. In this chapter we define models for *implementing* computational processes that achieve the desired relation, i.e., describe *how* to compute the output from the input. We will see several examples of such models using both Boolean circuits and straight-line programming languages.

- We can use *logical operations* such as *AND*, *OR*, and *NOT* to compute an output from an input (see Section 3.2).
- A *Boolean circuit* is a way to compose the basic logical operations to compute a more complex function (see Section 3.3). We can think of Boolean circuits as both a mathematical model (which is based on directed acyclic graphs) as well as physical devices we can construct in the real world in a variety of ways, including not just silicon-based semi-conductors but also mechanical and even biological mechanisms (see Section 3.5).
- We can describe Boolean circuits also as *straight-line programs*, which are programs that do not have any looping constructs (i.e., no `while` / `for`/ `do .. until` etc.), see Section 3.4.
- It is possible to implement the *AND*, *OR*, and *NOT* operations using the *NAND* operation (as well as vice versa). This means that circuits with *AND*/*OR*/*NOT* gates can compute the same functions (i.e., are *equivalent in power*) to circuits with *NAND* gates, and we can use either model to describe computation based on our convenience, see Section 3.6. To give out a "spoiler", we will see in Chapter 4 that such circuits can compute *all* finite functions.

One "big idea" of this chapter is the notion of *equivalence* between models (Big Idea 3). Two computational models are *equivalent* if they can compute the same set of functions.

Boolean circuits with *AND*/*OR*/*NOT* gates are equivalent to circuits with *NAND* gates, but this is just one example of the more general phenomenon that we will see many times in this book.

3.1 DEFINING COMPUTATION

The name "algorithm" is derived from the Latin transliteration of Muhammad ibn Musa al-Khwarizmi's name. Al-Khwarizmi was a Persian scholar during the 9th century whose books introduced the western world to the decimal positional numeral system, as well as to the solutions of linear and quadratic equations (see Fig. 3.4). However Al-Khwarizmi's descriptions of algorithms were rather informal by today's standards. Rather than use "variables" such as x, y, he used concrete numbers such as 10 and 39, and trusted the reader to be able to extrapolate from these examples, much as algorithms are still taught to children today.

Here is how Al-Khwarizmi described the algorithm for solving an equation of the form $x^2 + bx = c$:

> [*How to solve an equation of the form*] *"roots and squares are equal to numbers": For instance "one square, and ten roots of the same, amount to thirty-nine dirhems" that is to say, what must be the square which, when increased by ten of its own root, amounts to thirty-nine? The solution is this: you halve the number of the roots, which in the present instance yields five. This you multiply by itself; the product is twenty-five. Add this to thirty-nine' the sum is sixty-four. Now take the root of this, which is eight, and subtract from it half the number of roots, which is five; the remainder is three. This is the root of the square which you sought for; the square itself is nine.*

For the purposes of this book, we will need a much more precise way to describe algorithms. Fortunately (or is it unfortunately?), at least at the moment, computers lag far behind school-age children in learning from examples. Hence in the 20th century, people came up with exact formalisms for describing algorithms, namely *programming languages*. Here is al-Khwarizmi's quadratic equation solving algorithm described in the *Python* programming language:

```
from math import sqrt
#Pythonspeak to enable use of the sqrt function to compute
 ↪  square roots.

def solve_eq(b,c):
    # return solution of x^2 + bx = c following Al
     ↪  Khwarizmi's instructions
    # Al Kwarizmi demonstrates this for the case b=10 and
     ↪  c= 39
```

Figure 3.4: Text pages from Algebra manuscript with geometrical solutions to two quadratic equations. Shelfmark: MS. Huntington 214 fol. 004v-005r

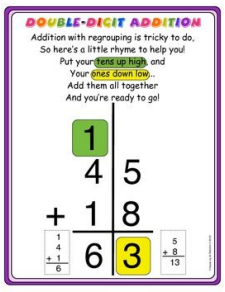


Figure 3.5: An explanation for children of the two digit addition algorithm

```
    val1 = b / 2.0 # "halve the number of the roots"
    val2 = val1 * val1 # "this you multiply by itself"
    val3 = val2 + c # "Add this to thirty-nine"
    val4 = sqrt(val3) # "take the root of this"
    val5 = val4 - val1 # "subtract from it half the number
     ↪  of roots"
    return val5  # "This is the root of the square which
     ↪  you sought for"

# Test: solve x^2 + 10*x = 39
print(solve_eq(10,39))
# 3.0
```

We can define algorithms informally as follows:

Informal definition of an algorithm: An *algorithm* is a set of instructions for how to compute an output from an input by following a sequence of "elementary steps".

An algorithm A *computes* a function F if for every input x, if we follow the instructions of A on the input x, we obtain the output $F(x)$.

In this chapter we will make this informal definition precise using the model of **Boolean Circuits**. We will show that Boolean Circuits are equivalent in power to **straight line programs** that are written in "ultra simple" programming languages that do not even have loops. We will also see that the particular choice of **elementary operations** is immaterial and many different choices yield models with equivalent power (see Fig. 3.6). However, it will take us some time to get there. We will start by discussing what are "elementary operations" and how we map a description of an algorithm into an actual physical process that produces an output from an input in the real world.

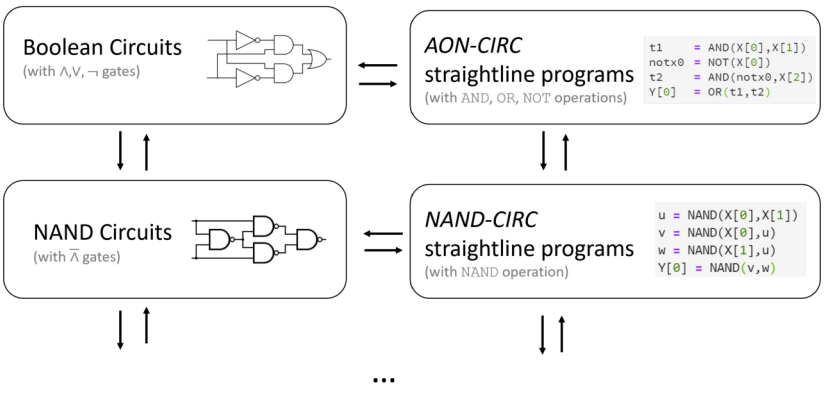


Figure 3.6: An overview of the computational models defined in this chapter. We will show several equivalent ways to represent a recipe for performing a finite computation. Specifically we will show that we can model such a computation using either a *Boolean circuit* or a *straight line program*, and these two representations are equivalent to one another. We will also show that we can choose as our basic operations either the set $\{AND, OR, NOT\}$ or the set $\{NAND\}$ and these two choices are equivalent in power. By making the choice of whether to use circuits or programs, and whether to use $\{AND, OR, NOT\}$ or $\{NAND\}$ we obtain four equivalent ways of modeling finite computation. Moreover, there are many other choices of sets of basic operations that are equivalent in power.

3.2 COMPUTING USING AND, OR, AND NOT.

An algorithm breaks down a *complex* calculation into a series of *simpler* steps. These steps can be executed in a variety of different ways, including:

- Writing down symbols on a piece of paper.
- Modifying the current flowing on electrical wires.
- Binding a protein to a strand of DNA.
- Responding to a stimulus by a member of a collection (e.g., a bee in a colony, a trader in a market).

To formally define algorithms, let us try to "err on the side of simplicity" and model our "basic steps" as truly minimal. For example, here are some very simple functions:

- $OR : \{0,1\}^2 \to \{0,1\}$ defined as

$$OR(a,b) = \begin{cases} 0 & a=b=0 \\ 1 & \text{otherwise} \end{cases}$$

- $AND : \{0,1\}^2 \to \{0,1\}$ defined as

$$AND(a,b) = \begin{cases} 1 & a=b=1 \\ 0 & \text{otherwise} \end{cases}$$

- $NOT : \{0,1\} \to \{0,1\}$ defined as

$$NOT(a) = \begin{cases} 0 & a=1 \\ 1 & a=0 \end{cases}$$

The functions *AND*, *OR* and *NOT*, are the basic logical operators used in logic and many computer systems. In the context of logic, it is common to use the notation $a \wedge b$ for $AND(a,b)$, $a \vee b$ for $OR(a,b)$ and $\overline{a}$ and $\neg a$ for $NOT(a)$, and we will use this notation as well.

Each one of the functions AND, OR, NOT takes either one or two single bits as input, and produces a single bit as output. Clearly, it cannot get much more basic than that. However, the power of computation comes from *composing* such simple building blocks together.

■ **Example 3.1 — Majority from AND,OR and NOT.** Consider the function $MAJ : \{0,1\}^3 \to \{0,1\}$ that is defined as follows:

$$MAJ(x) = \begin{cases} 1 & x_0 + x_1 + x_2 \geq 2 \\ 0 & \text{otherwise} \end{cases} .$$

That is, for every $x \in \{0,1\}^3$, $MAJ(x) = 1$ if and only if the majority (i.e., at least two out of the three) of x's elements are equal to 1. Can you come up with a formula involving *AND*, *OR* and *NOT* to compute *MAJ*? (It would be useful for you to pause at this point and work out the formula for yourself. As a hint, although the *NOT* operator is needed to compute some functions, you will not need to use it to compute *MAJ*.)

Let us first try to rephrase $MAJ(x)$ in words: "$MAJ(x) = 1$ if and only if there exists some pair of distinct elements i, j such that both x_i and x_j are equal to 1." In other words it means that $MAJ(x) = 1$ iff *either* both $x_0 = 1$ *and* $x_1 = 1$, *or* both $x_1 = 1$ *and* $x_2 = 1$, *or* both $x_0 = 1$ *and* $x_2 = 1$. Since the *OR* of three conditions c_0, c_1, c_2 can be written as $OR(c_0, OR(c_1, c_2))$, we can now translate this into a formula as follows:

$$MAJ(x_0, x_1, x_2) = OR\left(AND(x_0, x_1) \ , \ OR(AND(x_1, x_2) \ , \ AND(x_0, x_2)) \right) . \tag{3.1}$$

Recall that we can also write $a \vee b$ for $OR(a, b)$ and $a \wedge b$ for $AND(a, b)$. With this notation, (3.1) can also be written as

$$MAJ(x_0, x_1, x_2) = ((x_0 \wedge x_1) \vee (x_1 \wedge x_2)) \vee (x_0 \wedge x_2) .$$

We can also write (3.1) in a "programming language" form, expressing it as a set of instructions for computing *MAJ* given the basic operations *AND*, *OR*, *NOT*:

```
def MAJ(X[0],X[1],X[2]):
    firstpair  = AND(X[0],X[1])
    secondpair = AND(X[1],X[2])
    thirdpair  = AND(X[0],X[2])
    temp       = OR(secondpair,thirdpair)
    return OR(firstpair,temp)
```

3.2.1 Some properties of AND and OR

Like standard addition and multiplication, the functions *AND* and *OR* satisfy the properties of *commutativity*: $a \vee b = b \vee a$ and $a \wedge b = b \wedge a$ and *associativity*: $(a \vee b) \vee c = a \vee (b \vee c)$ and $(a \wedge b) \wedge c = a \wedge (b \wedge c)$. As in the case of addition and multiplication, we often drop the parenthesis

and write $a \vee b \vee c \vee d$ for $((a \vee b) \vee c) \vee d$, and similarly OR's and AND's of more terms. They also satisfy a variant of the distributive law:

Solved Exercise 3.1 — Distributive law for AND and OR. Prove that for every $a, b, c \in \{0,1\}$, $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.

Solution:

We can prove this by enumerating over all the 8 possible values for $a, b, c \in \{0,1\}$ but it also follows from the standard distributive law. Suppose that we identify any positive integer with "true" and the value zero with "false". Then for every numbers $u, v \in \mathbb{N}$, $u + v$ is positive if and only if $u \vee v$ is true and $u \cdot v$ is positive if and only if $u \wedge v$ is true. This means that for every $a, b, c \in \{0,1\}$, the expression $a \wedge (b \vee c)$ is true if and only if $a \cdot (b + c)$ is positive, and the expression $(a \wedge b) \vee (a \wedge c)$ is true if and only if $a \cdot b + a \cdot c$ is positive, But by the standard distributive law $a \cdot (b + c) = a \cdot b + a \cdot c$ and hence the former expression is true if and only if the latter one is.

3.2.2 Extended example: Computing *XOR* from *AND*, *OR*, and *NOT*

Let us see how we can obtain a different function from the same building blocks. Define $XOR : \{0,1\}^2 \to \{0,1\}$ to be the function $XOR(a, b) = a + b \mod 2$. That is, $XOR(0,0) = XOR(1,1) = 0$ and $XOR(1,0) = XOR(0,1) = 1$. We claim that we can construct *XOR* using only *AND*, *OR*, and *NOT*.

As usual, it is a good exercise to try to work out the algorithm for *XOR* using *AND*, *OR* and *NOT* on your own before reading further.

The following algorithm computes *XOR* using *AND*, *OR*, and *NOT*:

Algorithm 3.2 — XOR from $AND/OR/NOT$.

Input: $a, b \in \{0,1\}$.
Output: $XOR(a, b)$

1: $w1 \leftarrow AND(a, b)$
2: $w2 \leftarrow NOT(w1)$
3: $w3 \leftarrow OR(a, b)$
4: **return** $AND(w2, w3)$

Lemma 3.3 For every $a, b \in \{0,1\}$, on input a, b, Algorithm 3.2 outputs $a + b \mod 2$.

Proof. For every a, b, $XOR(a, b) = 1$ if and only if a is *different* from b. On input $a, b \in \{0, 1\}$, Algorithm 3.2 outputs $AND(w2, w3)$ where $w2 = NOT(AND(a, b))$ and $w3 = OR(a, b)$.

- If $a = b = 0$ then $w3 = OR(a, b) = 0$ and so the output will be 0.
- If $a = b = 1$ then $AND(a, b) = 1$ and so $w2 = NOT(AND(a, b)) = 0$ and the output will be 0.
- If $a = 1$ and $b = 0$ (or vice versa) then both $w3 = OR(a, b) = 1$ and $w1 = AND(a, b) = 0$, in which case the algorithm will output $OR(NOT(w1), w3) = 1$.

■

We can also express Algorithm 3.2 using a programming language. Specifically, the following is a *Python* program that computes the *XOR* function:

```
def AND(a,b): return a*b
def OR(a,b):  return 1-(1-a)*(1-b)
def NOT(a):   return 1-a

def XOR(a,b):
    w1 = AND(a,b)
    w2 = NOT(w1)
    w3 = OR(a,b)
    return AND(w2,w3)

# Test out the code
print([f"XOR({a},{b})={XOR(a,b)}" for a in [0,1] for b in
 ↪  [0,1]])
# ['XOR(0,0)=0', 'XOR(0,1)=1', 'XOR(1,0)=1', 'XOR(1,1)=0']
```

Solved Exercise 3.2 — Compute XOR on three bits of input. Let $XOR_3 : \{0,1\}^3 \to \{0,1\}$ be the function defined as $XOR_3(a, b, c) = a + b + c \mod 2$. That is, $XOR_3(a, b, c) = 1$ if $a+b+c$ is odd, and $XOR_3(a, b, c) = 0$ otherwise. Show that you can compute XOR_3 using AND, OR, and NOT. You can express it as a formula, use a programming language such as Python, or use a Boolean circuit.

■

Solution:

Addition modulo two satisfies the same properties of *associativity* ($(a + b) + c = a + (b + c)$) and *commutativity* ($a + b = b + a$) as standard addition. This means that, if we define $a \oplus b$ to equal $a + b$

mod 2, then

$$XOR_3(a,b,c) = (a \oplus b) \oplus c$$

or in other words

$$XOR_3(a,b,c) = XOR(XOR(a,b),c) \ .$$

Since we know how to compute *XOR* using AND, OR, and NOT, we can compose this to compute XOR_3 using the same building blocks. In Python this corresponds to the following program:

```
def XOR3(a,b,c):
    w1 = AND(a,b)
    w2 = NOT(w1)
    w3 = OR(a,b)
    w4 = AND(w2,w3)
    w5 = AND(w4,c)
    w6 = NOT(w5)
    w7 = OR(w4,c)
    return AND(w6,w7)

# Let's test this out
print([f"XOR3({a},{b},{c})={XOR3(a,b,c)}" for a in [0,1]
 ↪  for b in [0,1] for c in [0,1]])
# ['XOR3(0,0,0)=0', 'XOR3(0,0,1)=1', 'XOR3(0,1,0)=1',
 ↪  'XOR3(0,1,1)=0', 'XOR3(1,0,0)=1', 'XOR3(1,0,1)=0',
 ↪  'XOR3(1,1,0)=0', 'XOR3(1,1,1)=1']
```

Try to generalize the above examples to obtain a way to compute $XOR_n : \{0,1\}^n \to \{0,1\}$ for every n using at most $4n$ basic steps involving applications of a function in $\{AND, OR, NOT\}$ to outputs or previously computed values.

3.2.3 Informally defining "basic operations" and "algorithms"

We have seen that we can obtain at least some examples of interesting functions by composing together applications of *AND*, *OR*, and *NOT*. This suggests that we can use *AND*, *OR*, and *NOT* as our "basic operations", hence obtaining the following definition of an "algorithm":

Semi-formal definition of an algorithm: An *algorithm* consists of a sequence of steps of the form "compute a new value by applying *AND*, *OR*, or *NOT* to previously computed values".

An algorithm A *computes* a function F if for every input x to F, if we feed x as input to the algorithm, the value computed in its last step is $F(x)$.

There are several concerns that are raised by this definition:

1. First and foremost, this definition is indeed too informal. We do not specify exactly what each step does, nor what it means to "feed x as input".
2. Second, the choice of *AND*, *OR* or *NOT* seems rather arbitrary. Why not *XOR* and *MAJ*? Why not allow operations like addition and multiplication? What about any other logical constructions such `if/then` or `while`?
3. Third, do we even know that this definition has anything to do with actual computing? If someone gave us a description of such an algorithm, could we use it to actually compute the function in the real world?

These concerns will to a large extent guide us in the upcoming chapters. Thus you would be well advised to re-read the above informal definition and see what you think about these issues.

A large part of this book will be devoted to addressing the above issues. We will see that:

1. We can make the definition of an algorithm fully formal, and so give a precise mathematical meaning to statements such as "Algorithm A computes function f".
2. While the choice of *AND*/*OR*/*NOT* is arbitrary, and we could just as well have chosen other functions, we will also see this choice does not matter much. We will see that we would obtain the same computational power if we instead used addition and multiplication, and essentially every other operation that could be reasonably thought of as a basic step.
3. It turns out that we can and do compute such "*AND*/*OR*/*NOT*-based algorithms" in the real world. First of all, such an algorithm is clearly well specified, and so can be executed by a human with a pen and paper. Second, there are a variety of ways to *mechanize* this computation. We've already seen that we can write Python code that corresponds to following such a list of instructions. But in fact we can directly implement operations such as *AND*, *OR*, and *NOT*

via electronic signals using components known as *transistors*. This is how modern electronic computers operate.

In the remainder of this chapter, and the rest of this book, we will begin to answer some of these questions. We will see more examples of the power of simple operations to compute more complex operations including addition, multiplication, sorting and more. We will also discuss how to *physically implement* simple operations such as *AND*, *OR* and *NOT* using a variety of technologies.

3.3 BOOLEAN CIRCUITS

Boolean circuits provide a precise notion of "composing basic operations together". A Boolean circuit (see Fig. 3.9) is composed of *gates* and *inputs* that are connected by *wires*. The *wires* carry a signal that represents either the value 0 or 1. Each gate corresponds to either the *OR*, *AND*, or *NOT* operation. An *OR gate* has two incoming wires, and one or more outgoing wires. If these two incoming wires carry the signals a and b (for $a, b \in \{0,1\}$), then the signal on the outgoing wires will be $OR(a,b)$. *AND* and *NOT* gates are defined similarly. The *inputs* have only outgoing wires. If we set a certain input to a value $a \in \{0,1\}$, then this value is propagated on all the wires outgoing from it. We also designate some gates as *output gates*, and their value corresponds to the result of evaluating the circuit. For example, **??** gives such a circuit for the *XOR* function, following Section 3.2.2. We evaluate an n-input Boolean circuit C on an input $x \in \{0,1\}^n$ by placing the bits of x on the inputs, and then propagating the values on the wires until we reach an output, see Fig. 3.9.

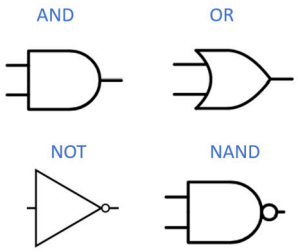


Figure 3.7: Standard symbols for the logical operations or "gates" of *AND*, *OR*, *NOT*, as well as the operation *NAND* discussed in Section 3.6.

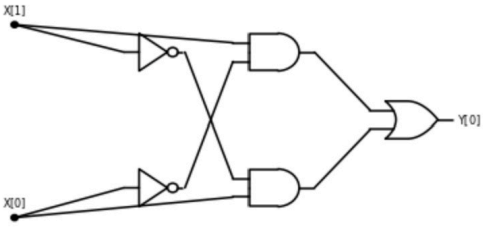


Figure 3.8: A circuit with *AND*, *OR* and *NOT* gates for computing the *XOR* function.

R

Remark 3.4 — Physical realization of Boolean circuits. Boolean circuits are a *mathematical model* that does not necessarily correspond to a physical object, but they can be implemented physically. In physical implementations of circuits, the signal is often implemented by electric potential, or *voltage*, on a wire, where for example voltage above a certain level is interpreted as a logical value of 1, and below a certain level is interpreted as a logical value of 0. Section 3.5 discusses physical implementations of Boolean circuits (with examples including using electrical signals such as in silicon-based circuits, as well as biological and mechanical implementations).

Solved Exercise 3.3 — All equal function. Define $ALLEQ : \{0,1\}^4 \to \{0,1\}$ to be the function that on input $x \in \{0,1\}^4$ outputs 1 if and only if $x_0 = x_1 = x_2 = x_3$. Give a Boolean circuit for computing *ALLEQ*.

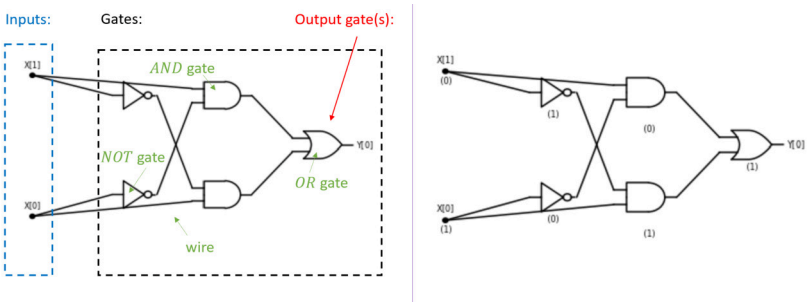


Figure 3.9: A *Boolean Circuit* consists of *gates* that are connected by *wires* to one another and the *inputs*. The left side depicts a circuit with 2 inputs and 5 gates, one of which is designated the output gate. The right side depicts the evaluation of this circuit on the input $x \in \{0,1\}^2$ with $x_0 = 1$ and $x_1 = 0$. The value of every gate is obtained by applying the corresponding function (*AND*, *OR*, or *NOT*) to values on the wire(s) that enter it. The output of the circuit on a given input is the value of the output gate(s). In this case, the circuit computes the *XOR* function and hence it outputs 1 on the input 10.

Solution:

Another way to describe the function *ALLEQ* is that it outputs 1 on an input $x \in \{0,1\}^4$ if and only if $x = 0^4$ or $x = 1^4$. We can phrase the condition $x = 1^4$ as $x_0 \wedge x_1 \wedge x_2 \wedge x_3$ which can be computed using three AND gates. Similarly we can phrase the condition $x = 0^4$ as $\overline{x}_0 \wedge \overline{x}_1 \wedge \overline{x}_2 \wedge \overline{x}_3$ which can be computed using four NOT gates and three AND gates. The output of *ALLEQ* is the OR of these two conditions, which results in the circuit of 4 NOT gates, 6 AND gates, and one OR gate presented in Fig. 3.10.

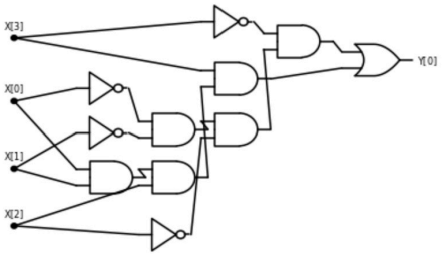


Figure 3.10: A Boolean circuit for computing the *all equal* function $ALLEQ : \{0,1\}^4 \rightarrow \{0,1\}$ that outputs 1 on $x \in \{0,1\}^4$ if and only if $x_0 = x_1 = x_2 = x_3$.

3.3.1 Boolean circuits: a formal definition

We defined Boolean circuits informally as obtained by connecting *AND*, *OR*, and *NOT* gates via wires so as to produce an output from an input. However, to be able to prove theorems about the existence or non-existence of Boolean circuits for computing various functions we need to:

1. Formally define a Boolean circuit as a mathematical object.
2. Formally define what it means for a circuit C to compute a function f.

We now proceed to do so. We will define a Boolean circuit as a labeled *Directed Acyclic Graph* (*DAG*). The *vertices* of the graph correspond to the gates and inputs of the circuit, and the *edges* of the graph correspond to the wires. A wire from an input or gate u to a gate v in the circuit corresponds to a directed edge between the corresponding vertices. The inputs are vertices with no incoming edges, while each gate has the appropriate number of incoming edges based on the function it computes. (That is, *AND* and *OR* gates have two in-neighbors, while *NOT* gates have one in-neighbor.) The formal definition is as follows (see also Fig. 3.11):

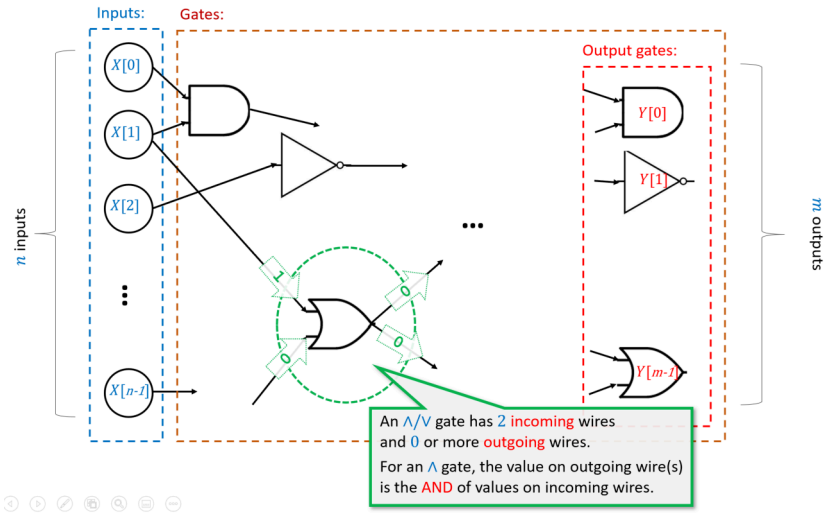


Figure 3.11: A *Boolean Circuit* is a labeled directed acyclic graph (DAG). It has n *input* vertices, which are marked with `X[0]`,..., `X[`$n-1$`]` and have no incoming edges, and the rest of the vertices are *gates*. *AND*, *OR*, and *NOT* gates have two, two, and one incoming edges, respectively. If the circuit has m outputs, then m of the gates are known as *outputs* and are marked with `Y[0]`,...,`Y[`$m-1$`]`. When we evaluate a circuit C on an input $x \in \{0,1\}^n$, we start by setting the value of the input vertices to $x_0, \ldots, x_{n-1}$ and then propagate the values, assigning to each gate g the result of applying the operation of g to the values of g's in-neighbors. The output of the circuit is the value assigned to the output gates.

Definition 3.5 — Boolean Circuits. Let n, m, s be positive integers with $s \geq m$. A *Boolean circuit* with n inputs, m outputs, and s gates, is a labeled directed acyclic graph (DAG) $G = (V, E)$ with $s+n$ vertices satisfying the following properties:

- Exactly n of the vertices have no in-neighbors. These vertices are known as *inputs* and are labeled with the n labels `X[0]`, ..., `X[`$n-1$`]`. Each input has at least one out-neighbor.
- The other s vertices are known as *gates*. Each gate is labeled with $\wedge$, $\vee$ or $\neg$. Gates labeled with $\wedge$ (*AND*) or $\vee$ (*OR*) have two in-neighbors. Gates labeled with $\neg$ (*NOT*) have one in-neighbor. We will allow parallel edges. [1]
- Exactly m of the gates are also labeled with the m labels `Y[0]`, ..., `Y[`$m-1$`]` (in addition to their label $\wedge/\vee/\neg$). These are known as *outputs*.

The *size* of a Boolean circuit is the number s of gates it contains.

P

This is a non-trivial mathematical definition, so it is worth taking the time to read it slowly and carefully. As in all mathematical definitions, we are using a known mathematical object — a directed acyclic graph (DAG) — to define a new object, a Boolean circuit. This might be a good time to review some of the basic

[1] Having parallel edges means an AND or OR gate u can have both its in-neighbors be the same gate v. Since $AND(a,a) = OR(a,a) = a$ for every $a \in \{0,1\}$, such parallel edges don't help in computing new values in circuits with AND/OR/NOT gates. However, we will see circuits with more general sets of gates later on.

properties of DAGs and in particular the fact that they can be *topologically sorted*, see Section 1.6.

If C is a circuit with n inputs and m outputs, and $x \in \{0,1\}^n$, then we can compute the output of C on the input x in the natural way: assign the input vertices `X[0]`, ..., `X[`$n-1$`]` the values $x_0, \ldots, x_{n-1}$, apply each gate on the values of its in-neighbors, and then output the values that correspond to the output vertices. Formally, this is defined as follows:

Definition 3.6 — Computing a function via a Boolean circuit. Let C be a Boolean circuit with n inputs and m outputs. For every $x \in \{0,1\}^n$, the *output* of C on the input x, denoted by $C(x)$, is defined as the result of the following process:

We let $h : V \to \mathbb{N}$ be the *minimal layering* of C (aka *topological sorting*, see Theorem 1.26). We let L be the maximum layer of h, and for $\ell = 0, 1, \ldots, L$ we do the following:

- For every v in the ℓ-th layer (i.e., v such that $h(v) = \ell$) do:
  - If v is an input vertex labeled with `X[`i`]` for some $i \in [n]$, then we assign to v the value x_i.
  - If v is a gate vertex labeled with $\wedge$ and with two in-neighbors u, w then we assign to v the *AND* of the values assigned to u and w. (Since u and w are in-neighbors of v, they are in a lower layer than v, and hence their values have already been assigned.)
  - If v is a gate vertex labeled with $\vee$ and with two in-neighbors u, w then we assign to v the OR of the values assigned to u and w.
  - If v is a gate vertex labeled with $\neg$ and with one in-neighbor u then we assign to v the negation of the value assigned to u.
- The result of this process is the value $y \in \{0,1\}^m$ such that for every $j \in [m]$, y_j is the value assigned to the vertex with label `Y[`j`]`.

Let $f : \{0,1\}^n \to \{0,1\}^m$. We say that the circuit C *computes* f if for every $x \in \{0,1\}^n$, $C(x) = f(x)$.

R

Remark 3.7 — Boolean circuits nitpicks (optional). In phrasing Definition 3.5, we've made some technical

choices that are not very important, but will be convenient for us later on. Having parallel edges means an AND or OR gate u can have both its in-neighbors be the same gate v. Since $AND(a,a) = OR(a,a) = a$ for every $a \in \{0,1\}$, such parallel edges don't help in computing new values in circuits with AND/OR/NOT gates. However, we will see circuits with more general sets of gates later on. The condition that every input vertex has at least one out-neighbor is also not very important because we can always add "dummy gates" that touch these inputs. However, it is convenient since it guarantees that (since every gate has at most two in-neighbors) the number of inputs in a circuit is never larger than twice its size.

3.4 STRAIGHT-LINE PROGRAMS

We have seen two ways to describe how to compute a function f using *AND*, *OR* and *NOT*:

- A *Boolean circuit*, defined in Definition 3.5, computes f by connecting via wires *AND*, *OR*, and *NOT* gates to the inputs.
- We can also describe such a computation using a *straight-line program* that has lines of the form `foo = AND(bar,blah)`, `foo = OR(bar,blah)` and `foo = NOT(bar)` where `foo`, `bar` and `blah` are variable names. (We call this a *straight-line program* since it contains no loops or branching (e.g., if/then) statements.)

To make the second definition more precise, we will now define a *programming language* that is equivalent to Boolean circuits. We call this programming language the **AON-CIRC programming language** ("AON" stands for *AND*/*OR*/*NOT*; "CIRC" stands for *circuit*).

For example, the following is an AON-CIRC program that on input $x \in \{0,1\}^2$, outputs $\overline{x_0 \wedge x_1}$ (i.e., the *NOT* operation applied to $AND(x_0, x_1)$):

```
temp = AND(X[0],X[1])
Y[0] = NOT(temp)
```

AON-CIRC is not a practical programming language: it was designed for pedagogical purposes only, as a way to model computation as the composition of *AND*, *OR*, and *NOT*. However, it can still be easily implemented on a computer.

Given this example, you might already be able to guess how to write a program for computing (for example) $x_0 \wedge \overline{x_1 \vee x_2}$, and in general how to translate a Boolean circuit into an AON-CIRC program. However, since we will want to prove mathematical statements about

AON-CIRC programs, we will need to precisely define the AON-CIRC programming language. Precise specifications of programming languages can sometimes be long and tedious,[2] but are crucial for secure and reliable implementations. Luckily, the AON-CIRC programming language is simple enough that we can define it formally with relatively little pain.

[2] For example the C programming language specification takes more than 500 pages.

3.4.1 Specification of the AON-CIRC programming language

An AON-CIRC program is a sequence of strings, which we call "lines", satisfying the following conditions:

- Every line has one of the following forms: `foo = AND(bar,baz)`, `foo = OR(bar,baz)`, or `foo = NOT(bar)` where `foo`, `bar` and `baz` are *variable identifiers*. (We follow the common programming languages convention of using names such as `foo`, `bar`, `baz` as stand-ins for generic identifiers.) The line `foo = AND(bar,baz)` corresponds to the operation of assigning to the variable `foo` the logical AND of the values of the variables `bar` and `baz`. Similarly `foo = OR(bar,baz)` and `foo = NOT(bar)` correspond to the logical OR and logical NOT operations.
- A *variable identifier* in the AON-CIRC programming language can be any combination of letters, numbers, underscores, and brackets. There are two special types of variables:
  - Variables of the form `X[`i`]`, with $i \in \{0, 1, \ldots, n-1\}$ are known as *input* variables.
  - Variables of the form `Y[`j`]` are known as *output* variables.
- A valid AON-CIRC program P includes input variables of the form `X[0]`,...,`X[`$n-1$`]` and output variables of the form `Y[0]`,..., `Y[`$m-1$`]` where n, m are natural numbers. We say that n is the number of *inputs* of the program P and m is the number of outputs.
- In a valid AON-CIRC program, in every line the variables on the right-hand side of the assignment operator must either be input variables or variables that have already been assigned a value in a previous line.
- If P is a valid AON-CIRC program of n inputs and m outputs, then for every $x \in \{0,1\}^n$ the *output* of P on input x is the string $y \in \{0,1\}^m$ defined as follows:
  - Initialize the input variables `X[0]`,...,`X[`$n-1$`]` to the values $x_0, \ldots, x_{n-1}$
  - Run the operator lines of P one by one in order, in each line assigning to the variable on the left-hand side of the assignment operators the value of the operation on the right-hand side.

  - Let $y \in \{0,1\}^m$ be the values of the output variables `Y[0]`,..., `Y[`$m-1$`]` at the end of the execution.

- We denote the output of P on input x by $P(x)$.
- The *size* of an AON circ program P is the number of lines it contains. (The reader might note that this corresponds to our definition of the size of a circuit as the number of gates it contains.)

Now that we formally specified AON-CIRC programs, we can define what it means for an AON-CIRC program P to compute a function f:

Definition 3.8 — Computing a function via AON-CIRC programs. Let $f : \{0,1\}^n \to \{0,1\}^m$, and P be a valid AON-CIRC program with n inputs and m outputs. We say that P *computes* f if $P(x) = f(x)$ for every $x \in \{0,1\}^n$.

The following solved exercise gives an example of an AON-CIRC program.

Solved Exercise 3.4 Consider the following function $CMP : \{0,1\}^4 \to \{0,1\}$ that on four input bits $a, b, c, d \in \{0,1\}$, outputs 1 iff the number represented by (a, b) is larger than the number represented by (c, d). That is $CMP(a, b, c, d) = 1$ iff $2a + b > 2c + d$.

Write an AON-CIRC program to compute CMP.

■

Solution:

Writing such a program is tedious but not truly hard. To compare two numbers we first compare their most significant digit, and then go down to the next digit and so on and so forth. In this case where the numbers have just two binary digits, these comparisons are particularly simple. The number represented by (a, b) is larger than the number represented by (c, d) if and only if one of the following conditions happens:

1. The most significant bit a of (a, b) is larger than the most significant bit c of (c, d).

   or

2. The two most significant bits a and c are equal, but $b > d$.

Another way to express the same condition is the following: the number (a, b) is larger than (c, d) iff $a > c$ **OR** ($a \geq c$ **AND** $b > d$).

For binary digits α, β, the condition $\alpha > \beta$ is simply that $\alpha = 1$ and $\beta = 0$ or $AND(\alpha, NOT(\beta)) = 1$, and the condition $\alpha \geq \beta$ is simply $OR(\alpha, NOT(\beta)) = 1$. Together these observations can be used to give the following AON-CIRC program to compute *CMP*:

```
# Compute CMP:{0,1}^4-->{0,1}
# CMP(X)=1 iff 2X[0]+X[1] > 2X[2] + X[3]
temp_1 = NOT(X[2])
temp_2 = AND(X[0],temp_1)
temp_3 = OR(X[0],temp_1)
temp_4 = NOT(X[3])
temp_5 = AND(X[1],temp_4)
temp_6 = AND(temp_5,temp_3)
Y[0] = OR(temp_2,temp_6)
```

We can also present this 8-line program as a circuit with 8 gates, see Fig. 3.12.

■

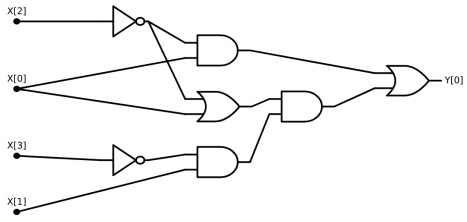


Figure 3.12: A circuit for computing the *CMP* function. The evaluation of this circuit on $(1, 1, 1, 0)$ yields the output 1, since the number 3 (represented in binary as 11) is larger than the number 2 (represented in binary as 10).

3.4.2 Proving equivalence of AON-CIRC programs and Boolean circuits

We now formally prove that AON-CIRC programs and Boolean circuits have exactly the same power:

Theorem 3.9 — Equivalence of circuits and straight-line programs. Let $f : \{0,1\}^n \to \{0,1\}^m$ and $s \geq m$ be some number. Then f is computable by a Boolean circuit with s gates if and only if f is computable by an AON-CIRC program of s lines.

Proof Idea:

The idea is simple - AON-CIRC programs and Boolean circuits are just different ways of describing the exact same computational process. For example, an *AND* gate in a Boolean circuit corresponds to computing the *AND* of two previously-computed values. In an AON-CIRC program this will correspond to the line that stores in a variable the `AND` of two previously-computed variables.

⋆

P

This proof of Theorem 3.9 is simple at heart, but all the details it contains can make it a little cumbersome to read. You might be better off trying to work it out yourself before reading it. Our GitHub repository contains a "proof by Python" of Theorem 3.9: implementation of functions `circuit2prog` and `prog2circuits` mapping Boolean circuits to AON-CIRC programs and vice versa.

Proof of Theorem 3.9. Let $f : \{0,1\}^n \to \{0,1\}^m$. Since the theorem is an "if and only if" statement, to prove it we need to show both directions: translating an AON-CIRC program that computes f into a circuit that computes f, and translating a circuit that computes f into an AON-CIRC program that does so.

We start with the first direction. Let P be an AON-CIRC program that computes f. We define a circuit C as follows: the circuit will have n inputs and s gates. For every $i \in [s]$, if the i-th operator line has the form `foo = AND(bar,blah)` then the i-th gate in the circuit will be an AND gate that is connected to gates j and k where j and k correspond to the last lines before i where the variables `bar` and `blah` (respectively) were written to. (For example, if $i = 57$ and the last line `bar` was written to is 35 and the last line `blah` was written to is 17 then the two in-neighbors of gate 57 will be gates 35 and 17.) If either `bar` or `blah` is an input variable then we connect the gate to the corresponding input vertex instead. If `foo` is an output variable of the form `Y[`j`]` then we add the same label to the corresponding gate to mark it as an output gate. We do the analogous operations if the i-th line involves an `OR` or a `NOT` operation (except that we use the corresponding *OR* or *NOT* gate, and in the latter case have only one in-neighbor instead of two). For every input $x \in \{0,1\}^n$, if we run the program P on x, then the value written that is computed in the i-th line is exactly the value that will be assigned to the i-th gate if we evaluate the circuit C on x. Hence $C(x) = P(x)$ for every $x \in \{0,1\}^n$.

For the other direction, let C be a circuit of s gates and n inputs that computes the function f. We sort the gates according to a topological order and write them as $v_0, \ldots, v_{s-1}$. We now can create a program P of s operator lines as follows. For every $i \in [s]$, if v_i is an AND gate with in-neighbors v_j, v_k then we will add a line to P of the form `temp_`i `= AND(temp_`j`,temp_`k`)`, unless one of the vertices is an input vertex or an output gate, in which case we change this to the form `X[.]` or `Y[.]` appropriately. Because we work in topological ordering, we are guaranteed that the in-neighbors v_j and v_k correspond to variables that have already been assigned a value. We do the same for OR and NOT gates. Once again, one can verify that for every input x, the value $P(x)$ will equal $C(x)$ and hence the program computes the same function as the circuit. (Note that since C is a valid circuit, per Definition 3.5, every input vertex of C has at least one out-neighbor and there are exactly m output gates labeled $0, \ldots, m-1$; hence all the variables `X[0]`, ..., `X[`$n-1$`]` and `Y[0]` ,..., `Y[`$m-1$`]` will appear in the program P.)

■

```
temp_1 = AND(X[0],X[1])
temp_2 = NOT(temp_1)
temp_3 = OR(X[0],X[1])
temp_4 = AND(temp_2,temp_3)
temp_5 = AND(temp_4,X[2])
temp_6 = NOT(temp_5)
temp_7 = OR(temp_4,X[2])
Y[0] = AND(temp_6,temp_7)
```

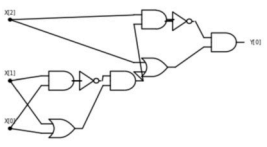


Figure 3.13: Two equivalent descriptions of the same AND/OR/NOT computation as both an AON program and a Boolean circuit.

3.5 PHYSICAL IMPLEMENTATIONS OF COMPUTING DEVICES (DIGRESSION)

Computation is an abstract notion that is distinct from its physical *implementations*. While most modern computing devices are obtained by mapping logical gates to semiconductor-based transistors, throughout history people have computed using a huge variety of mechanisms, including mechanical systems, gas and liquid (known as *fluidics*), biological and chemical processes, and even living creatures (e.g., see Fig. 3.14 or this video for how crabs or slime mold can be used to do computations).

In this section we will review some of these implementations, both so you can get an appreciation of how it is possible to directly translate Boolean circuits to the physical world, without going through the entire stack of architecture, operating systems, and compilers, as well as to emphasize that silicon-based processors are by no means the only way to perform computation. Indeed, as we will see in Chapter 23, a very exciting recent line of work involves using different media for computation that would allow us to take advantage of *quantum mechanical effects* to enable different types of algorithms.

> Such a cool way to explain logic gates. pic.twitter.com/6Wgu2ZKFCx
> — Lionel Page (@page_eco) October 28, 2019

3.5.1 Transistors

A *transistor* can be thought of as an electric circuit with two inputs, known as the *source* and the *gate* and an output, known as the *sink*. The gate controls whether current flows from the source to the sink. In a *standard transistor*, if the gate is "ON" then current can flow from the source to the sink and if it is "OFF" then it can't. In a *complementary transistor* this is reversed: if the gate is "OFF" then current can flow from the source to the sink and if it is "ON" then it can't.

There are several ways to implement the logic of a transistor. For example, we can use faucets to implement it using water pressure (e.g. Fig. 3.15). This might seem as merely a curiosity, but there is a field known as fluidics concerned with implementing logical operations using liquids or gasses. Some of the motivations include operating in extreme environmental conditions such as in space or a battlefield, where standard electronic equipment would not survive.

The standard implementations of transistors use electrical current. One of the original implementations used *vacuum tubes*. As its name implies, a vacuum tube is a tube containing nothing (i.e., a vacuum) and where a priori electrons could freely flow from the source (a wire) to the sink (a plate). However, there is a gate (a grid) between the two, where modulating its voltage can block the flow of electrons.

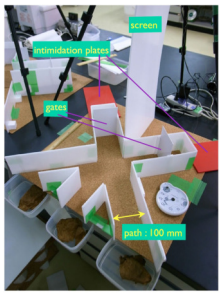


Figure 3.14: Crab-based logic gates from the paper "Robust soldier-crab ball gate" by Gunji, Nishiyama and Adamatzky. This is an example of an AND gate that relies on the tendency of two swarms of crabs arriving from different directions to combine to a single swarm that continues in the average of the directions.

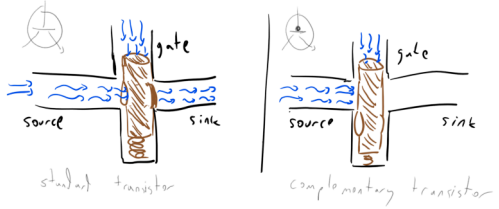


Figure 3.15: We can implement the logic of transistors using water. The water pressure from the gate closes or opens a faucet between the source and the sink.

Early vacuum tubes were roughly the size of lightbulbs (and looked very much like them too). In the 1950's they were supplanted by *transistors*, which implement the same logic using *semiconductors* which are materials that normally do not conduct electricity but whose conductivity can be modified and controlled by inserting impurities ("doping") and applying an external electric field (this is known as the *field effect*). In the 1960's computers started to be implemented using *integrated circuits* which enabled much greater density. In 1965, Gordon Moore predicted that the number of transistors per integrated circuit would double every year (see Fig. 3.16), and that this would lead to "such wonders as home computers —or at least terminals connected to a central computer— automatic controls for automobiles, and personal portable communications equipment". Since then, (adjusted versions of) this so-called "Moore's law" have been running strong, though exponential growth cannot be sustained forever, and some physical limitations are already becoming apparent.

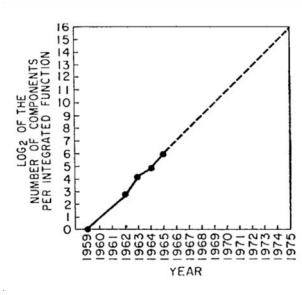


Figure 3.16: The number of transistors per integrated circuit from 1959 till 1965 and a prediction that exponential growth will continue for at least another decade. Figure taken from "Cramming More Components onto Integrated Circuits", Gordon Moore, 1965

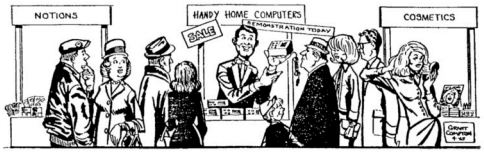


Figure 3.17: Cartoon from Gordon Moore's article "predicting" the implications of radically improving transistor density.

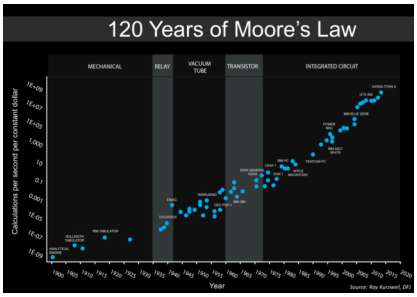


Figure 3.18: The exponential growth in computing power over the last 120 years. Graph by Steve Jurvetson, extending a prior graph of Ray Kurzweil.

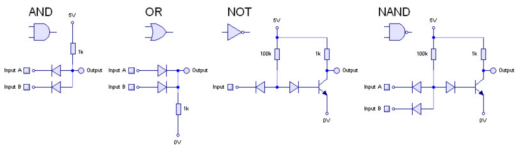


Figure 3.19: Implementing logical gates using transistors. Figure taken from Rory Mangles' website.

3.5.2 Logical gates from transistors

We can use transistors to implement various Boolean functions such as *AND*, *OR*, and *NOT*. For each a two-input gate $G : \{0,1\}^2 \to \{0,1\}$, such an implementation would be a system with two input wires x, y and one output wire z, such that if we identify high voltage with "1" and low voltage with "0", then the wire z will be equal to "1" if and only if applying G to the values of the wires x and y is 1 (see Fig. 3.19 and Fig. 3.20). This means that if there exists a AND/OR/NOT circuit to compute a function $g : \{0,1\}^n \to \{0,1\}^m$, then we can compute g in the physical world using transistors as well.

3.5.3 Biological computing

Computation can be based on biological or chemical systems. For example the *lac* operon produces the enzymes needed to digest lactose only if the conditions $x \wedge (\neg y)$ hold where x is "lactose is present" and y is "glucose is present". Researchers have managed to create transistors, and from them logic gates, based on DNA molecules (see also Fig. 3.21). Projects such as the Cello programming language enable converting Boolean circuits into DNA sequences that encode operations that can be executed in bacterial cells, see this video. One motivation for DNA computing is to achieve increased parallelism or storage density; another is to create "smart biological agents" that could perhaps be injected into bodies, replicate themselves, and fix or kill cells that were damaged by a disease such as cancer. Computing in biological systems is not restricted, of course, to DNA: even larger systems such as flocks of birds can be considered as computational processes.

3.5.4 Cellular automata and the game of life

Cellular automata is a model of a system composed of a sequence of *cells*, each of which can have a finite state. At each step, a cell updates its state based on the states of its *neighboring cells* and some simple rules. As we will discuss later in this book (see Section 8.4), cellular automata such as Conway's "Game of Life" can be used to simulate computation gates.

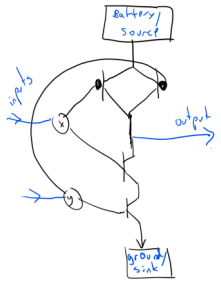


Figure 3.20: Implementing a NAND gate (see Section 3.6) using transistors.

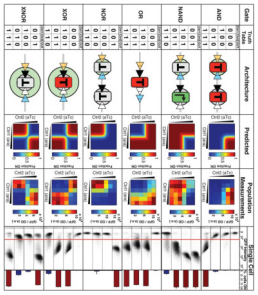


Figure 3.21: Performance of DNA-based logic gates. Figure taken from paper of Bonnet et al, Science, 2013.

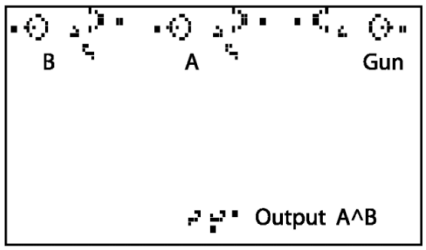


Figure 3.22: An AND gate using a "Game of Life" configuration. Figure taken from Jean-Philippe Rennard's paper.

3.5.5 Neural networks

One computation device that we all carry with us is our own *brain*. Brains have served humanity throughout history, doing computations that range from distinguishing prey from predators, through making scientific discoveries and artistic masterpieces, to composing witty 280 character messages. The exact working of the brain is still not fully understood, but one common mathematical model for it is a (very large) *neural network*.

A neural network can be thought of as a Boolean circuit that instead of *AND*/*OR*/*NOT* uses some other gates as the basic basis. For example, one particular basis we can use are *threshold gates*. For every vector $w = (w_0, \ldots, w_{k-1})$ of integers and integer t (some or all of which could be negative), the *threshold function corresponding to* w, t is the function $T_{w,t} : \{0,1\}^k \to \{0,1\}$ that maps $x \in \{0,1\}^k$ to 1 if and only if $\sum_{i=0}^{k-1} w_i x_i \geq t$. For example, the threshold function $T_{w,t}$ corresponding to $w = (1,1,1,1,1)$ and $t = 3$ is simply the majority function MAJ_5 on $\{0,1\}^5$. Threshold gates can be thought of as an approximation for *neuron cells* that make up the core of human and animal brains. To a first approximation, a neuron has k inputs and a single output, and the neuron "fires" or "turns on" its output when those signals pass some threshold.

Many machine learning algorithms use *artificial neural networks* whose purpose is not to imitate biology but rather to perform some computational tasks, and hence are not restricted to a threshold or other biologically-inspired gates. Generally, a neural network is often described as operating on signals that are real numbers, rather than $0/1$ values, and where the output of a gate on inputs $x_0, \ldots, x_{k-1}$ is obtained by applying $f(\sum_i w_i x_i)$ where $f : \mathbb{R} \to \mathbb{R}$ is an activation function such as rectified linear unit (ReLU), Sigmoid, or many others (see Fig. 3.23). However, for the purposes of our discussion, all of the above are equivalent (see also Exercise 3.13). In particular we can reduce the setting of real inputs to binary inputs by representing a real number in the binary basis, and multiplying the weight of the bit corresponding to the ith digit by 2^i.

3.5.6 A computer made from marbles and pipes

We can implement computation using many other physical media, without any electronic, biological, or chemical components. Many suggestions for *mechanical* computers have been put forward, going back at least to Gottfried Leibniz's computing machines from the 1670s and Charles Babbage's 1837 plan for a mechanical "Analytical Engine". As one example, Fig. 3.24 shows a simple implementation of a NAND (negation of AND, see Section 3.6) gate using marbles going through pipes. We represent a logical value in $\{0,1\}$ by a pair of pipes, such that there is a marble flowing through exactly one of the pipes. We call one of the pipes the "0 pipe" and the other the "1 pipe", and so the identity of the pipe containing the marble determines the logical value. A NAND gate corresponds to a mechanical object with two pairs of incoming pipes and one pair of outgoing pipes, such that for every $a,b \in \{0,1\}$, if two marbles are rolling toward the object in the a pipe of the first pair and the b pipe of the second pair, then a marble will roll out of the object in the $NAND(a,b)$-pipe of the outgoing pair. In fact, there is even a commercially-available educational game that uses marbles as a basis of computing, see Fig. 3.26.

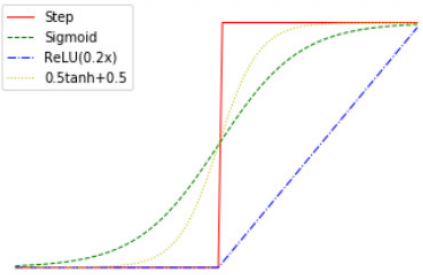


Figure 3.23: Common activation functions used in Neural Networks, including rectified linear units (ReLU), sigmoids, and hyperbolic tangent. All of those can be thought of as continuous approximations to simplify the step function. All of these can be used to compute the NAND gate (see Exercise 3.13). This property enables neural networks to (approximately) compute any function that can be computed by a Boolean circuit.

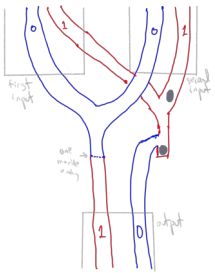


Figure 3.24: A physical implementation of a NAND gate using marbles. Each wire in a Boolean circuit is modeled by a pair of pipes representing the values 0 and 1 respectively, and hence a gate has four input pipes (two for each logical input) and two output pipes. If one of the input pipes representing the value 0 has a marble in it then that marble will flow to the output pipe representing the value 1. (The dashed line represents a gadget that will ensure that at most one marble is allowed to flow onward in the pipe.) If both the input pipes representing the value 1 have marbles in them, then the first marble will be stuck but the second one will flow onwards to the output pipe representing the value 0.

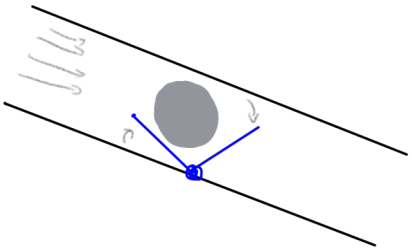


Figure 3.25: A "gadget" in a pipe that ensures that at most one marble can pass through it. The first marble that passes causes the barrier to lift and block new ones.

3.6 THE NAND FUNCTION

The *NAND* function is another simple function that is extremely useful for defining computation. It is the function mapping $\{0,1\}^2$ to $\{0,1\}$ defined by:

$$NAND(a,b) = \begin{cases} 0 & a=b=1 \\ 1 & \text{otherwise} \end{cases} .$$

As its name implies, *NAND* is the NOT of AND (i.e., $NAND(a,b) = NOT(AND(a,b))$), and so we can clearly compute *NAND* using *AND* and *NOT*. Interestingly, the opposite direction holds as well:

Theorem 3.10 — NAND computes AND,OR,NOT. We can compute *AND*, *OR*, and *NOT* by composing only the *NAND* function.

Proof. We start with the following observation. For every $a \in \{0,1\}$, $AND(a,a) = a$. Hence, $NAND(a,a) = NOT(AND(a,a)) = NOT(a)$. This means that *NAND* can compute *NOT*. By the principle of "double negation", $AND(a,b) = NOT(NOT(AND(a,b)))$, and hence we can use *NAND* to compute *AND* as well. Once we can compute *AND* and *NOT*, we can compute *OR* using "De Morgan's Law": $OR(a,b) = NOT(AND(NOT(a),NOT(b)))$ (which can also be written as $a \vee b = \overline{\overline{a} \wedge \overline{b}}$) for every $a,b \in \{0,1\}$. ■

P

Theorem 3.10's proof is very simple, but you should make sure that **(i)** you understand the statement of the theorem, and **(ii)** you follow its proof. In particular, you should make sure you understand why De Morgan's law is true.

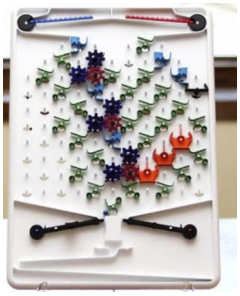


Figure 3.26: The game "Turing Tumble" contains an implementation of logical gates using marbles.

We can use *NAND* to compute many other functions, as demonstrated in the following exercise.

Solved Exercise 3.5 — Compute majority with NAND. Let $MAJ : \{0,1\}^3 \to \{0,1\}$ be the function that on input a, b, c outputs 1 iff $a + b + c \geq 2$. Show how to compute *MAJ* using a composition of *NAND*'s.

■

Solution:

Recall that (3.1) states that

$$MAJ(x_0, x_1, x_2) = OR\left(\, AND(x_0, x_1)\ ,\ OR(AND(x_1, x_2)\ ,\ AND(x_0, x_2))\,\right)\ . \tag{3.2}$$

We can use Theorem 3.10 to replace all the occurrences of *AND* and *OR* with *NAND*'s. Specifically, we can use the equivalence $AND(a,b) = NOT(NAND(a,b))$, $OR(a,b) = NAND(NOT(a), NOT(b))$, and $NOT(a) = NAND(a,a)$ to replace the right-hand side of (3.2) with an expression involving only *NAND*, yielding that $MAJ(a,b,c)$ is equivalent to the (somewhat unwieldy) expression

$$NAND\Big(\, NAND\big(\, NAND(NAND(a,b), NAND(a,c)),\ NAND(NAND(a,b), NAND(a,c))\,\big),\ NAND(b,c)\,\Big)$$

The same formula can also be expressed as a circuit with NAND gates, see Fig. 3.27.

■

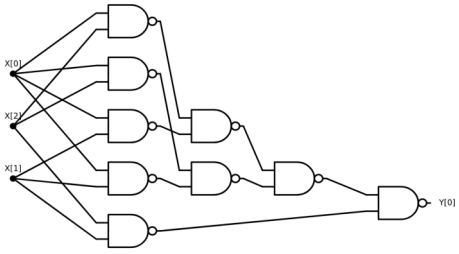


Figure 3.27: A circuit with NAND gates to compute the Majority function on three bits

3.6.1 NAND Circuits

We define *NAND Circuits* as circuits in which all the gates are NAND operations. Such a circuit again corresponds to a directed acyclic graph (DAG) since all the gates correspond to the same function (i.e., NAND), we do not even need to label them, and all gates have in-degree exactly two. Despite their simplicity, NAND circuits can be quite powerful.

■ **Example 3.11 — $NAND$ circuit for XOR.** Recall the *XOR* function which maps $x_0, x_1 \in \{0,1\}$ to $x_0 + x_1 \mod 2$. We have seen in Section 3.2.2 that we can compute *XOR* using *AND*, *OR*, and *NOT*, and so by Theorem 3.10 we can compute it using only *NAND*'s. However, the following is a direct construction of computing *XOR* by a sequence of NAND operations:

1. Let $u = NAND(x_0, x_1)$.
2. Let $v = NAND(x_0, u)$
3. Let $w = NAND(x_1, u)$.
4. The *XOR* of x_0 and x_1 is $y_0 = NAND(v, w)$.

One can verify that this algorithm does indeed compute *XOR* by enumerating all the four choices for $x_0, x_1 \in \{0,1\}$. We can also represent this algorithm graphically as a circuit, see Fig. 3.28.

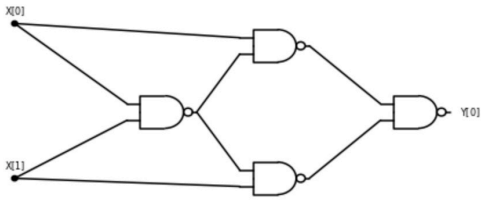


Figure 3.28: A circuit with NAND gates to compute the XOR of two bits.

In fact, we can show the following theorem:

Theorem 3.12 — NAND is a universal operation. For every Boolean circuit C of s gates, there exists a NAND circuit C' of at most $3s$ gates that computes the same function as C.

Proof Idea:

The idea of the proof is to just replace every *AND*, *OR* and *NOT* gate with their NAND implementation following the proof of Theorem 3.10.

⋆

Proof of Theorem 3.12. If C is a Boolean circuit, then since, as we've seen in the proof of Theorem 3.10, for every $a, b \in \{0,1\}$

- $NOT(a) = NAND(a, a)$
- $AND(a, b) = NAND(NAND(a, b), NAND(a, b))$
- $OR(a, b) = NAND(NAND(a, a), NAND(b, b))$

we can replace every gate of C with at most three *NAND* gates to obtain an equivalent circuit C'. The resulting circuit will have at most $3s$ gates.

■

Big Idea 3 Two models are *equivalent in power* if they can be used to compute the same set of functions.

3.6.2 More examples of NAND circuits (optional)

Here are some more sophisticated examples of NAND circuits:

Incrementing integers. Consider the task of computing, given as input a string $x \in \{0,1\}^n$ that represents a natural number $X \in \mathbb{N}$, the representation of $X+1$. That is, we want to compute the function $INC_n : \{0,1\}^n \to \{0,1\}^{n+1}$ such that for every $x_0, \ldots, x_{n-1}$, $INC_n(x) = y$ which satisfies $\sum_{i=0}^{n} y_i 2^i = \left(\sum_{i=0}^{n-1} x_i 2^i\right) + 1$. (For simplicity of notation, in this example we use the representation where the least significant digit is first rather than last.)

The increment operation can be very informally described as follows: *"Add 1 to the least significant bit and propagate the carry"*. A little more precisely, in the case of the binary representation, to obtain the increment of x, we scan x from the least significant bit onwards, and flip all 1's to 0's until we encounter a bit equal to 0, in which case we flip it to 1 and stop.

Thus we can compute the increment of $x_0, \ldots, x_{n-1}$ by doing the following:

Algorithm 3.13 — Compute Increment Function.

Input: $x_0, x_1, \ldots, x_{n-1}$ representing the number $\sum_{i=0}^{n-1} x_i \cdot 2^i$
 # we use LSB-first representation
Output: $y \in \{0,1\}^{n+1}$ such that $\sum_{i=0}^{n} y_i \cdot 2^i = \sum_{i=0}^{n-1} x_i \cdot 2^i + 1$

```
Let c_0 ← 1        # we pretend we have a "carry" of 1 initially
for i = 0, ..., n − 1 do
    Let y_i ← XOR(x_i, c_i).
    if c_i = x_i = 1 then
        c_{i+1} = 1
    else
        c_{i+1} = 0
    end if
end for
Let y_n ← c_n.
```

Algorithm 3.13 describes precisely how to compute the increment operation, and can be easily transformed into *Python* code that performs the same computation, but it does not seem to directly yield a NAND circuit to compute this. However, we can transform this algorithm line by line to a NAND circuit. For example, since for every a, $NAND(a, NOT(a)) = 1$, we can replace the initial statement $c_0 = 1$ with $c_0 = NAND(x_0, NAND(x_0, x_0))$. We already know how to compute *XOR* using NAND and so we can use this to implement the operation $y_i \leftarrow XOR(x_i, c_i)$. Similarly, we can write the "if" statement as saying $c_{i+1} \leftarrow AND(c_i, x_i)$, or in other words

$c_{i+1} \leftarrow NAND(NAND(c_i, x_i), NAND(c_i, x_i))$. Finally, the assignment $y_n = c_n$ can be written as $y_n = NAND(NAND(c_n, c_n), NAND(c_n, c_n))$. Combining these observations yields for every $n \in \mathbb{N}$, a *NAND* circuit to compute INC_n. For example, Fig. 3.29 shows what this circuit looks like for $n = 4$.

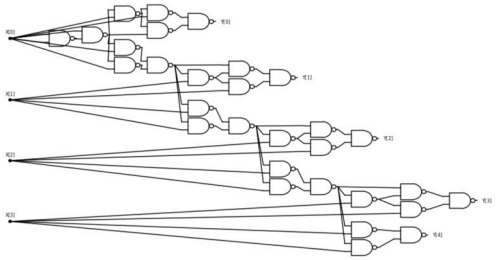


Figure 3.29: NAND circuit with computing the *increment* function on 4 bits.

From increment to addition. Once we have the increment operation, we can certainly compute addition by repeatedly incrementing (i.e., compute $x+y$ by performing $INC(x)$ y times). However, that would be quite inefficient and unnecessary. With the same idea of keeping track of carries we can implement the "grade-school" addition algorithm and compute the function $ADD_n : \{0,1\}^{2n} \rightarrow \{0,1\}^{n+1}$ that on input $x \in \{0,1\}^{2n}$ outputs the binary representation of the sum of the numbers represented by $x_0, \ldots, x_{n-1}$ and $x_n, \ldots, x_{2n-1}$:

Algorithm 3.14 — Addition using NAND.

Input: $u \in \{0,1\}^n, v \in \{0,1\}^n$ representing numbers in LSB-first binary representation.
Output: LSB-first binary representation of $x + y$.

Let $c_0 \leftarrow 0$
for $i = 0, \ldots, n-1$ **do**
    Let $y_i \leftarrow u_i + v_i \mod 2$
    if $u_i + v_i + c_i \geq 2$ **then**
        $c_{i+1} \leftarrow 1$
    else
        $c_{i+1} \leftarrow 0$
    end if
end for
Let $y_n \leftarrow c_n$

Once again, Algorithm 3.14 can be translated into a NAND circuit. The crucial observation is that the "if/then" statement simply corresponds to $c_{i+1} \leftarrow MAJ_3(u_i, v_i, v_i)$ and we have seen in Solved Exercise 3.5 that the function $MAJ_3 : \{0,1\}^3 \rightarrow \{0,1\}$ can be computed using *NAND*s.

3.6.3 The NAND-CIRC Programming language

Just like we did for Boolean circuits, we can define a programming-language analog of NAND circuits. It is even simpler than the AON-CIRC language since we only have a single operation. We define the *NAND-CIRC Programming Language* to be a programming language where every line (apart from the input/output declaration) has the following form:

```
foo = NAND(bar,blah)
```

where `foo`, `bar` and `blah` are variable identifiers.

■ **Example 3.15 — Our first NAND-CIRC program.** Here is an example of a NAND-CIRC program:

```
u = NAND(X[0],X[1])
v = NAND(X[0],u)
w = NAND(X[1],u)
Y[0] = NAND(v,w)
```

P Do you know what function this program computes?
Hint: you have seen it before.

Formally, just like we did in Definition 3.8 for AON-CIRC, we can define the notion of computation by a NAND-CIRC program in the natural way:

Definition 3.16 — Computing by a NAND-CIRC program. Let $f : \{0,1\}^n \to \{0,1\}^m$ be some function, and let P be a NAND-CIRC program. We say that P *computes* the function f if:

1. P has n input variables `X[0]`, ... ,`X[`$n-1$`]` and m output variables `Y[0]`,...,`Y[`$m-1$`]`.
2. For every $x \in \{0,1\}^n$, if we execute P when we assign to `X[0]`, ... ,`X[`$n-1$`]` the values $x_0, \ldots, x_{n-1}$, then at the end of the execution, the output variables `Y[0]`,...,`Y[`$m-1$`]` have the values $y_0, \ldots, y_{m-1}$ where $y = f(x)$.

As before we can show that NAND circuits are equivalent to NAND-CIRC programs (see Fig. 3.30):

Theorem 3.17 — NAND circuits and straight-line program equivalence. For every $f : \{0,1\}^n \to \{0,1\}^m$ and $s \geq m$, f is computable by a NAND-CIRC program of s lines if and only if f is computable by a NAND circuit of s gates.

We omit the proof of Theorem 3.17 since it follows along exactly the same lines as the equivalence of Boolean circuits and AON-CIRC program (Theorem 3.9). Given Theorem 3.17 and Theorem 3.12, we know that we can translate every s-line AON-CIRC program P into an equivalent NAND-CIRC program of at most $3s$ lines. In fact, this translation can be easily done by replacing every line of the form

```
temp_1 = NAND(X[0],X[1])
temp_2 = NAND(X[0],temp_1)
temp_3 = NAND(X[1],temp_1)
temp_4 = NAND(temp_2,temp_3)
temp_5 = NAND(temp_4,X[2])
temp_6 = NAND(temp_4,temp_5)
temp_7 = NAND(X[2],temp_5)
Y[0] = NAND(temp_6,temp_7)
```

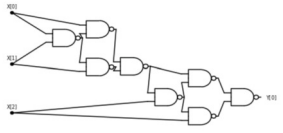


Figure 3.30: A NAND program and the corresponding circuit. Note how every line in the program corresponds to a gate in the circuit.

`foo = AND(bar,blah)`, `foo = OR(bar,blah)` or `foo = NOT(bar)` with the equivalent 1-3 lines that use the `NAND` operation. Our GitHub repository contains a "proof by code": a simple Python program `AON2NAND` that transforms an AON-CIRC into an equivalent NAND-CIRC program.

Remark 3.18 — Is the NAND-CIRC programming language Turing Complete? (optional note). You might have heard of a term called "Turing Complete" that is sometimes used to describe programming languages. (If you haven't, feel free to ignore the rest of this remark: we define this term precisely in Chapter 8.) If so, you might wonder if the NAND-CIRC programming language has this property. The answer is **no**, or perhaps more accurately, the term "Turing Completeness" is not really applicable for the NAND-CIRC programming language. The reason is that, by design, the NAND-CIRC programming language can only compute *finite* functions $F : \{0,1\}^n \to \{0,1\}^m$ that take a fixed number of input bits and produce a fixed number of outputs bits. The term "Turing Complete" is only applicable to programming languages for *infinite* functions that can take inputs of arbitrary length. We will come back to this distinction later on in this book.

3.7 EQUIVALENCE OF ALL THESE MODELS

If we put together Theorem 3.9, Theorem 3.12, and Theorem 3.17, we obtain the following result:

Theorem 3.19 — Equivalence between models of finite computation. For every sufficiently large s, n, m and $f : \{0,1\}^n \to \{0,1\}^m$, the following conditions are all equivalent to one another:

- f can be computed by a Boolean circuit (with $\wedge, \vee, \neg$ gates) of at most $O(s)$ gates.
- f can be computed by an AON-CIRC straight-line program of at most $O(s)$ lines.
- f can be computed by a NAND circuit of at most $O(s)$ gates.
- f can be computed by a NAND-CIRC straight-line program of at most $O(s)$ lines.

By "$O(s)$" we mean that the bound is at most $c \cdot s$ where c is a constant that is independent of n. For example, if f can be computed by a Boolean circuit of s gates, then it can be computed by a NAND-CIRC

program of at most $3s$ lines, and if f can be computed by a NAND circuit of s gates, then it can be computed by an AON-CIRC program of at most $2s$ lines.

Proof Idea:

We omit the formal proof, which is obtained by combining Theorem 3.9, Theorem 3.12, and Theorem 3.17. The key observation is that the results we have seen allow us to translate a program/circuit that computes f in one of the above models into a program/circuit that computes f in another model by increasing the lines/gates by at most a constant factor (in fact this constant factor is at most 3).

⋆

Theorem 3.9 is a special case of a more general result. We can consider even more general models of computation, where instead of AND/OR/NOT or NAND, we use other operations (see Section 3.7.1 below). It turns out that Boolean circuits are equivalent in power to such models as well. The fact that all these different ways to define computation lead to equivalent models shows that we are "on the right track". It justifies the seemingly arbitrary choices that we've made of using AND/OR/NOT or NAND as our basic operations, since these choices do not affect the power of our computational model. Equivalence results such as Theorem 3.19 mean that we can easily translate between Boolean circuits, NAND circuits, NAND-CIRC programs and the like. We will use this ability later on in this book, often shifting to the most convenient formulation without making a big deal about it. Hence we will not worry too much about the distinction between, for example, Boolean circuits and NAND-CIRC programs.

In contrast, we will continue to take special care to distinguish between *circuits/programs* and *functions* (recall Big Idea 2). A function corresponds to a *specification* of a computational task, and it is a fundamentally different object than a program or a circuit, which corresponds to the *implementation* of the task.

3.7.1 Circuits with other gate sets

There is nothing special about AND/OR/NOT or NAND. For every set of functions $\mathcal{G} = \{G_0, \ldots, G_{k-1}\}$, we can define a notion of circuits that use elements of $\mathcal{G}$ as gates, and a notion of a "$\mathcal{G}$ programming language" where every line involves assigning to a variable `foo` the result of applying some $G_i \in \mathcal{G}$ to previously defined or input variables. Specifically, we can make the following definition:

Definition 3.20 — General straight-line programs. Let $\mathcal{F} = \{f_0, \ldots, f_{t-1}\}$ be a finite collection of Boolean functions, such that $f_i : \{0,1\}^{k_i} \rightarrow$

$\{0,1\}$ for some $k_i \in \mathbb{N}$. An $\mathcal{F}$ *program* is a sequence of lines, each of which assigns to some variable the result of applying some $f_i \in \mathcal{F}$ to k_i other variables. As above, we use `X[`i`]` and `Y[`j`]` to denote the input and output variables.

We say that $\mathcal{F}$ is a *universal set of operations* (also known as a universal gate set) if there exists a $\mathcal{F}$ program to compute the function *NAND*.

AON-CIRC programs correspond to $\{AND, OR, NOT\}$ programs, NAND-CIRC programs corresponds to $\mathcal{F}$ programs for the set $\mathcal{F}$ that only contains the *NAND* function, but we can also define $\{IF, ZERO, ONE\}$ programs (see below), or use any other set.

We can also define $\mathcal{F}$ *circuits*, which will be directed graphs in which each *gate* corresponds to applying a function $f_i \in \mathcal{F}$, and will each have k_i incoming wires and a single outgoing wire. (If the function f_i is not *symmetric*, in the sense that the order of its input matters then we need to label each wire entering a gate as to which parameter of the function it corresponds to.) As in Theorem 3.9, we can show that $\mathcal{F}$ circuits and $\mathcal{F}$ programs are equivalent. We have seen that for $\mathcal{F} = \{AND, OR, NOT\}$, the resulting circuits/programs are equivalent in power to the NAND-CIRC programming language, as we can compute *NAND* using *AND*/*OR*/*NOT* and vice versa. This turns out to be a special case of a general phenomenon — the *universality* of *NAND* and other gate sets — that we will explore more in-depth later in this book.

■ **Example 3.21 — IF,ZERO,ONE circuits.** Let $\mathcal{F} = \{IF, ZERO, ONE\}$ where $ZERO : \{0,1\} \rightarrow \{0\}$ and $ONE : \{0,1\} \rightarrow \{1\}$ are the constant zero and one functions, [3] and $IF : \{0,1\}^3 \rightarrow \{0,1\}$ is the function that on input (a,b,c) outputs b if $a = 1$ and c otherwise. Then $\mathcal{F}$ is universal.

Indeed, we can demonstrate that $\{IF, ZERO, ONE\}$ is universal using the following formula for *NAND*:

$$NAND(a,b) = IF(a, IF(b, ZERO, ONE), ONE) \ .$$

[3] One can also define these functions as taking a length zero input. This makes no difference for the computational power of the model.

There are also some sets $\mathcal{F}$ that are more restricted in power. For example it can be shown that if we use only AND or OR gates (without NOT) then we do *not* get an equivalent model of computation. The exercises cover several examples of universal and non-universal gate sets.

3.7.2 Specification vs. implementation (again)

As we discussed in Section 2.6.1, one of the most important distinctions in this book is that of *specification* versus *implementation* or sep-

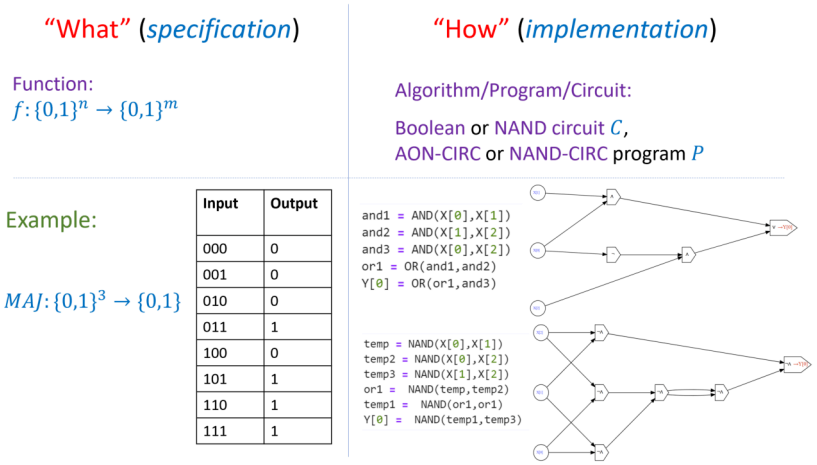


Figure 3.31: It is crucial to distinguish between the *specification* of a computational task, namely *what* is the function that is to be computed and the *implementation* of it, namely the algorithm, program, or circuit that contains the instructions defining *how* to map an input to an output. The same function could be computed in many different ways.

arating "what" from "how" (see Fig. 3.31). A *function* corresponds to the *specification* of a computational task, that is *what* output should be produced for every particular input. A *program* (or circuit, or any other way to specify *algorithms*) corresponds to the *implementation* of *how* to compute the desired output from the input. That is, a program is a set of instructions on how to compute the output from the input. Even within the same computational model there can be many different ways to compute the same function. For example, there is more than one NAND-CIRC program that computes the majority function, more than one Boolean circuit to compute the addition function, and so on and so forth.

Confusing specification and implementation (or equivalently *functions* and *programs*) is a common mistake, and one that is unfortunately encouraged by the common programming-language terminology of referring to parts of programs as "functions". However, in both the theory and practice of computer science, it is important to maintain this distinction, and it is particularly important for us in this book.

Chapter Recap

- An *algorithm* is a recipe for performing a computation as a sequence of "elementary" or "simple" operations.
- One candidate definition for "elementary" operations is the set *AND*, *OR* and *NOT*.
- Another candidate definition for an "elementary" operation is the *NAND* operation. It is an operation that is easily implementable in the physical world in a variety of methods including by electronic transistors.

- We can use *NAND* to compute many other functions, including majority, increment, and others.
- There are other equivalent choices, including the sets $\{AND, OR, NOT\}$ and $\{IF, ZERO, ONE\}$.
- We can formally define the notion of a function $F : \{0,1\}^n \to \{0,1\}^m$ being computable using the *NAND-CIRC Programming language*.
- For every set of basic operations, the notions of being computable by a circuit and being computable by a straight-line program are equivalent.

3.8 EXERCISES

Exercise 3.1 — Compare 4 bit numbers. Give a Boolean circuit (with AND/OR/NOT gates) that computes the function $CMP_8 : \{0,1\}^8 \to \{0,1\}$ such that $CMP_8(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3) = 1$ if and only if the number represented by $a_0a_1a_2a_3$ is larger than the number represented by $b_0b_1b_2b_3$.

■

Exercise 3.2 — Compare n bit numbers. Prove that there exists a constant c such that for every n there is a Boolean circuit (with AND/OR/NOT gates) C of at most $c \cdot n$ gates that computes the function $CMP_{2n} : \{0,1\}^{2n} \to \{0,1\}$ such that $CMP_{2n}(a_0 \cdots a_{n-1} b_0 \cdots b_{n-1}) = 1$ if and only if the number represented by $a_0 \cdots a_{n-1}$ is larger than the number represented by $b_0 \cdots b_{n-1}$.

■

Exercise 3.3 — OR,NOT is universal. Prove that the set $\{OR, NOT\}$ is *universal*, in the sense that one can compute NAND using these gates.

■

Exercise 3.4 — AND,OR is not universal. Prove that for every n-bit input circuit C that contains only AND and OR gates, as well as gates that compute the constant functions 0 and 1, C is *monotone*, in the sense that if $x, x' \in \{0,1\}^n$, $x_i \leq x'_i$ for every $i \in [n]$, then $C(x) \leq C(x')$.

Conclude that the set $\{AND, OR, 0, 1\}$ is *not* universal.

■

Exercise 3.5 — XOR is not universal. Prove that for every n-bit input circuit C that contains only XOR gates, as well as gates that compute the constant functions 0 and 1, C is *affine or linear modulo two*, in the sense that there exists some $a \in \{0,1\}^n$ and $b \in \{0,1\}$ such that for every $x \in \{0,1\}^n$, $C(x) = \sum_{i=0}^{n-1} a_i x_i + b \mod 2$.

Conclude that the set $\{XOR, 0, 1\}$ is *not* universal.

■

Exercise 3.6 — MAJ,NOT, 1 is universal. Let $MAJ : \{0,1\}^3 \to \{0,1\}$ be the majority function. Prove that $\{MAJ, NOT, 1\}$ is a universal set of gates.

Exercise 3.7 — MAJ,NOT is not universal. Prove that $\{MAJ, NOT\}$ is not a universal set. See footnote for hint.[4]

Exercise 3.8 — NOR is universal. Let $NOR : \{0,1\}^2 \to \{0,1\}$ defined as $NOR(a,b) = NOT(OR(a,b))$. Prove that $\{NOR\}$ is a universal set of gates.

Exercise 3.9 — Lookup is universal. Prove that $\{LOOKUP_1, 0, 1\}$ is a universal set of gates where 0 and 1 are the constant functions and $LOOKUP_1 : \{0,1\}^3 \to \{0,1\}$ satisfies $LOOKUP_1(a,b,c)$ equals a if $c = 0$ and equals b if $c = 1$.

Exercise 3.10 — Bound on universal basis size (challenge). Prove that for every subset B of the functions from $\{0,1\}^k$ to $\{0,1\}$, if B is universal then there is a B-circuit of at most $O(1)$ gates to compute the $NAND$ function (you can start by showing that there is a B circuit of at most $O(k^{16})$ gates).[5]

Exercise 3.11 — Size and inputs / outputs. Prove that for every NAND circuit of size s with n inputs and m outputs, $s \geq \min\{n/2, m\}$. See footnote for hint.[6]

Exercise 3.12 — Threshold using NANDs. Prove that there is some constant c such that for every $n > 1$, and integers $a_0, \ldots, a_{n-1}, b \in \{-2^n, -2^n + 1, \ldots, -1, 0, +1, \ldots, 2^n\}$, there is a NAND circuit with at most cn^4 gates that computes the *threshold* function $f_{a_0,\ldots,a_{n-1},b} : \{0,1\}^n \to \{0,1\}$ that on input $x \in \{0,1\}^n$ outputs 1 if and only if $\sum_{i=0}^{n-1} a_i x_i > b$.

Exercise 3.13 — NANDs from activation functions. We say that a function $f : \mathbb{R}^2 \to \mathbb{R}$ is a *NAND approximator* if it has the following property: for every $a, b \in \mathbb{R}$, if $\min\{|a|, |1-a|\} \leq 1/3$ and $\min\{|b|, |1-b|\} \leq 0.1$ then $|f(a,b) - NAND(\lfloor a \rceil, \lfloor b \rceil)| \leq 0.1$ where we denote by $\lfloor x \rceil$ the integer closest to x. That is, if a, b are within a distance $1/3$ to $\{0,1\}$ then we want $f(a,b)$ to equal the $NAND$ of the values in $\{0,1\}$ that are closest to a and b respectively. Otherwise, we do not care what the output of f is on a and b.

In this exercise you will show that you can construct a NAND approximator from many common activation functions used in deep

[4] *Hint:* Use the fact that $MAJ(\overline{a}, \overline{b}, \overline{c}) = \overline{MAJ(a,b,c)}$ to prove that every $f : \{0,1\}^n \to \{0,1\}$ computable by a circuit with only MAJ and NOT gates satisfies $f(0,0,\ldots,0) \neq f(1,1,\ldots,1)$. Thanks to Nathan Brunelle and David Evans for suggesting this exercise.

[5] Thanks to Alec Sun and Simon Fischer for comments on this problem.

[6] *Hint:* Use the conditions of Definition 3.5 stipulating that every input vertex has at least one out-neighbor and there are exactly m output gates. See also Remark 3.7.

neural networks. As a corollary you will obtain that deep neural networks can simulate NAND circuits. Since NAND circuits can also simulate deep neural networks, these two computational models are equivalent to one another.

1. Show that there is a NAND approximator f defined as $f(a,b) = L(DReLU(L'(a,b)))$ where $L' : \mathbb{R}^2 \to \mathbb{R}$ is an *affine* function (of the form $L'(a,b) = \alpha a + \beta b + \gamma$ for some $\alpha, \beta, \gamma \in \mathbb{R}$), L is an affine function (of the form $L(y) = \alpha y + \beta$ for $\alpha, \beta \in \mathbb{R}$), and $DReLU : \mathbb{R} \to \mathbb{R}$, is the function defined as $DReLU(x) = \min(1, \max(0, x))$. Note that $DReLU(x) = 1 - ReLU(1 - ReLU(x))$ where $ReLU(x) = \max(x, 0)$ is the rectified linear unit activation function.

2. Show that there is a NAND approximator f defined as $f(a,b) = L(sigmoid(L'(a,b)))$ where L', L are affine as above and $sigmoid : \mathbb{R} \to \mathbb{R}$ is the function defined as $sigmoid(x) = e^x/(e^x + 1)$.

3. Show that there is a NAND approximator f defined as $f(a,b) = L(tanh(L'(a,b)))$ where L', L are affine as above and $tanh : \mathbb{R} \to \mathbb{R}$ is the function defined as $tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$.

4. Prove that for every NAND-circuit C with n inputs and one output that computes a function $g : \{0,1\}^n \to \{0,1\}$, if we replace every gate of C with a NAND-approximator and then invoke the resulting circuit on some $x \in \{0,1\}^n$, the output will be a number y such that $|y - g(x)| \leq 1/3$.

■

Exercise 3.14 — Majority with NANDs efficiently. Prove that there is some constant c such that for every $n > 1$, there is a NAND circuit of at most $c \cdot n$ gates that computes the majority function on n input bits $MAJ_n : \{0,1\}^n \to \{0,1\}$. That is $MAJ_n(x) = 1$ iff $\sum_{i=0}^{n-1} x_i > n/2$. See footnote for hint.[7]

[7] One approach to solve this is using recursion and analyzing it using the so called "Master Theorem".

■

Exercise 3.15 — Output at last layer. Prove that for every $f : \{0,1\}^n \to \{0,1\}$, if there is a Boolean circuit C of s gates that computes f then there is a Boolean circuit C' of at most s gates such that in the minimal layering of C', the output gate of C' is placed in the last layer. See footnote for hint.[8]

[8] *Hint:* Vertices in layers beyond the output can be safely removed without changing the functionality of the circuit.

■

3.9 BIOGRAPHICAL NOTES

The excerpt from Al-Khwarizmi's book is from "The Algebra of Ben-Musa", Fredric Rosen, 1831.

Charles Babbage (1791-1871) was a visionary scientist, mathematician, and inventor (see [Swa02; CM00]). More than a century before the invention of modern electronic computers, Babbage realized that computation can be in principle mechanized. His first design for a mechanical computer was the *difference engine* that was designed to do polynomial interpolation. He then designed the *analytical engine* which was a much more general machine and the first prototype for a programmable general-purpose computer. Unfortunately, Babbage was never able to complete the design of his prototypes. One of the earliest people to realize the engine's potential and far-reaching implications was Ada Lovelace (see the notes for Chapter 7).

Boolean algebra was first investigated by Boole and DeMorgan in the 1840's [Boo47; De 47]. The definition of Boolean circuits and connection to electrical relay circuits was given in Shannon's Masters Thesis [Sha38]. (Howard Gardener called Shannon's thesis "possibly the most important, and also the most famous, master's thesis of the [20th] century".) Savage's book [Sav98], like this one, introduces the theory of computation starting with Boolean circuits as the first model. Jukna's book [Juk12] contains a modern in-depth exposition of Boolean circuits, see also [Weg87].

The NAND function was shown to be universal by Sheffer [She13], though this also appears in the earlier work of Peirce, see [Bur78]. Whitehead and Russell used NAND as the basis for their logic in their magnum opus *Principia Mathematica* [WR12]. In her Ph.D thesis, Ernst [Ern09] investigates empirically the minimal NAND circuits for various functions. Nisan and Shocken's book [NS05] builds a computing system starting from NAND gates and ending with high-level programs and games ("NAND to Tetris"); see also the website nandtotetris.org.

We defined the *size* of a Boolean circuit in Definition 3.5 to be the number of gates it contains. This is one of two conventions used in the literature. The other convention is to define the size as the number of *wires* (equivalent to the number of gates plus the number of inputs). This makes very little difference in almost all settings, but can affect the circuit size complexity of some "pathological examples" of functions such as the constant zero function that do not depend on much of their inputs.

Learning Objectives:

- Get comfortable with syntactic sugar or automatic translation of higher-level logic to low-level gates.
- Learn proof of major result: every finite function can be computed by a Boolean circuit.
- Start thinking *quantitatively* about the number of lines required for computation.

4

Syntactic sugar, and computing every function

"[In 1951] I had a running compiler and nobody would touch it because, they carefully told me, computers could only do arithmetic; they could not do programs.", Grace Murray Hopper, 1986.

"Syntactic sugar causes cancer of the semicolon.", Alan Perlis, 1982.

The computational models we considered thus far are as "bare bones" as they come. For example, our NAND-CIRC "programming language" has only the single operation `foo = NAND(bar,blah)`. In this chapter we will see that these simple models are actually *equivalent* to more sophisticated ones. The key observation is that we can implement more complex features using our basic building blocks, and then use these new features themselves as building blocks for even more sophisticated features. This is known as "syntactic sugar" in the field of programming language design since we are not modifying the underlying programming model itself, but rather we merely implement new features by syntactically transforming a program that uses such features into one that doesn't.

This chapter provides a "toolkit" that can be used to show that many functions can be computed by NAND-CIRC programs, and hence also by Boolean circuits. We will also use this toolkit to prove a fundamental theorem: *every* finite function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a Boolean circuit, see Theorem 4.13 below. While the syntactic sugar toolkit is important in its own right, Theorem 4.13 can also be proven directly without using this toolkit. We present this alternative proof in Section 4.5. See Fig. 4.1 for an outline of the results of this chapter.

This chapter: A non-mathy overview

In this chapter, we will see our first major result: *every* finite function can be computed by some Boolean circuit (see Theorem 4.13 and Big Idea 5). This is sometimes known as

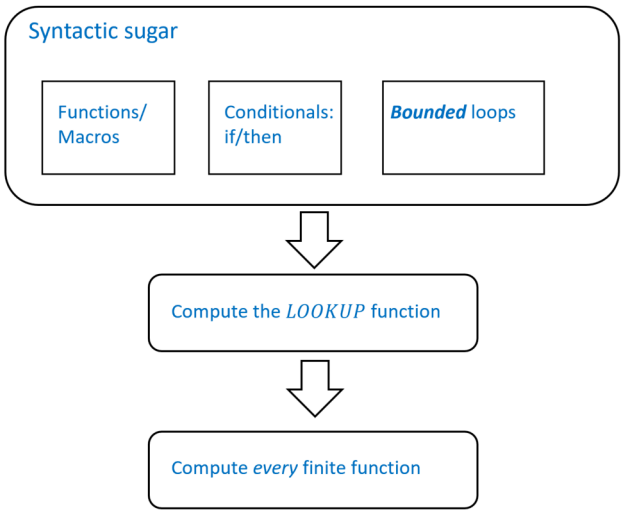


Figure 4.1: An outline of the results of this chapter. In Section 4.1 we give a toolkit of "syntactic sugar" transformations showing how to implement features such as programmer-defined functions and conditional statements in NAND-CIRC. We use these tools in Section 4.3 to give a NAND-CIRC program (or alternatively a Boolean circuit) to compute the *LOOKUP* function. We then build on this result to show in Section 4.4 that NAND-CIRC programs (or equivalently, Boolean circuits) can compute *every* finite function. An alternative direct proof of the same result is given in Section 4.5.

the "universality" of *AND*, *OR*, and *NOT* (and, using the equivalence of Chapter 3, of *NAND* as well)

Despite being an important result, Theorem 4.13 is actually not that hard to prove. Section 4.5 presents a relatively simple direct proof of this result. However, in Section 4.1 and Section 4.3 we derive this result using the concept of "syntactic sugar" (see Big Idea 4). This is an important concept for programming languages theory and practice. The idea behind "syntactic sugar" is that we can extend a programming language by implementing advanced features from its basic components. For example, we can take the AON-CIRC and NAND-CIRC programming languages we saw in Chapter 3, and extend them to achieve features such as user-defined functions (e.g., `def Foo(...)`), conditional statements (e.g., `if blah ...`), and more. Once we have these features, it is not that hard to show that we can take the "truth table" (table of all inputs and outputs) of any function, and use that to create an AON-CIRC or NAND-CIRC program that maps each input to its corresponding output.

We will also get our first glimpse of *quantitative measures* in this chapter. While Theorem 4.13 tells us that every function can be computed by *some* circuit, the number of gates in this circuit can be exponentially large. (We are not using here "exponentially" as some colloquial term for "very very big" but in a very precise mathematical sense, which also happens to coincide with being very very big.) It turns out that *some functions* (for example, integer addition and multi-

plication) can be in fact computed using far fewer gates. We will explore this issue of "gate complexity" more deeply in Chapter 5 and following chapters.

4.1 SOME EXAMPLES OF SYNTACTIC SUGAR

We now present some examples of "syntactic sugar" transformations that we can use in constructing straightline programs or circuits. We focus on the *straight-line programming language* view of our computational models, and specifically (for the sake of concreteness) on the NAND-CIRC programming language. This is convenient because many of the syntactic sugar transformations we present are easiest to think about in terms of applying "search and replace" operations to the source code of a program. However, by Theorem 3.19, all of our results hold equally well for circuits, whether ones using NAND gates or Boolean circuits that use the AND, OR, and NOT operations. Enumerating the examples of such syntactic sugar transformations can be a little tedious, but we do it for two reasons:

1. To convince you that despite their seeming simplicity and limitations, simple models such as Boolean circuits or the NAND-CIRC programming language are actually quite powerful.

2. So you can realize how lucky you are to be taking a theory of computation course and not a compilers course... :)

4.1.1 User-defined procedures

One staple of almost any programming language is the ability to define and then execute *procedures* or *subroutines*. (These are often known as *functions* in some programming languages, but we prefer the name *procedures* to avoid confusion with the function that a program computes.) The NAND-CIRC programming language does not have this mechanism built in. However, we can achieve the same effect using the time-honored technique of "copy and paste". Specifically, we can replace code which defines a procedure such as

```
def Proc(a,b):
    proc_code
    return c
some_code
f = Proc(d,e)
some_more_code
```

with the following code where we "paste" the code of `Proc`

```
some_code
proc_code'
some_more_code
```

and where `proc_code'` is obtained by replacing all occurrences of `a` with `d`, `b` with `e`, and `c` with `f`. When doing that we will need to ensure that all other variables appearing in `proc_code'` don't interfere with other variables. We can always do so by renaming variables to new names that were not used before. The above reasoning leads to the proof of the following theorem:

Theorem 4.1 — Procedure definition syntatic sugar. Let NAND-CIRC-PROC be the programming language NAND-CIRC augmented with the syntax above for defining procedures. Then for every NAND-CIRC-PROC program P, there exists a standard (i.e., "sugar-free") NAND-CIRC program P' that computes the same function as P.

R

Remark 4.2 — No recursive procedure. NAND-CIRC-PROC only allows *non-recursive* procedures. In particular, the code of a procedure `Proc` cannot call `Proc` but only use procedures that were defined before it. Without this restriction, the above "search and replace" procedure might never terminate and Theorem 4.1 would not be true.

Theorem 4.1 can be proven using the transformation above, but since the formal proof is somewhat long and tedious, we omit it here.

■ **Example 4.3 — Computing Majority from NAND using syntactic sugar.** Procedures allow us to express NAND-CIRC programs much more cleanly and succinctly. For example, because we can compute AND, OR, and NOT using NANDs, we can compute the *Majority* function as follows:

```
def NOT(a):
    return NAND(a,a)
def AND(a,b):
    temp = NAND(a,b)
    return NOT(temp)
def OR(a,b):
    temp1 = NOT(a)
    temp2 = NOT(b)
```

```
    return NAND(temp1,temp2)

def MAJ(a,b,c):
    and1 = AND(a,b)
    and2 = AND(a,c)
    and3 = AND(b,c)
    or1 = OR(and1,and2)
    return OR(or1,and3)

print(MAJ(0,1,1))
# 1
```

Fig. 4.2 presents the "sugar-free" NAND-CIRC program (and the corresponding circuit) that is obtained by "expanding out" this program, replacing the calls to procedures with their definitions.

Big Idea 4 Once we show that a computational model X is equivalent to a model that has feature Y, we can assume we have Y when showing that a function f is computable by X.

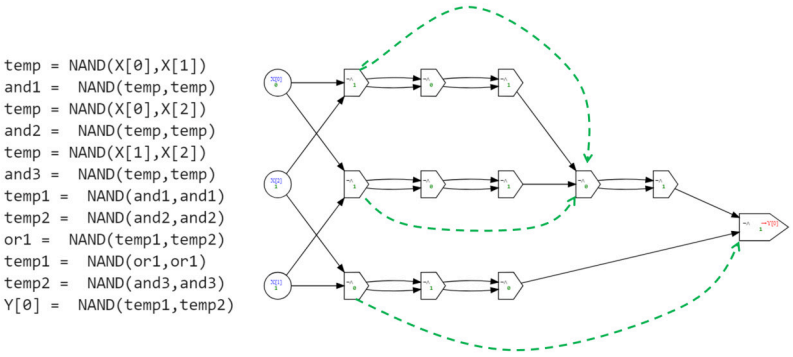


Figure 4.2: A standard (i.e., "sugar-free") NAND-CIRC program that is obtained by expanding out the procedure definitions in the program for Majority of Example 4.3. The corresponding circuit is on the right. Note that this is not the most efficient NAND circuit/program for majority: we can save on some gates by "shortcutting" steps where a gate u computes $NAND(v,v)$ and then a gate w computes $NAND(u,u)$ (as indicated by the dashed green arrows in the above figure).

R

Remark 4.4 — Counting lines. While we can use syntactic sugar to *present* NAND-CIRC programs in more readable ways, we did not change the definition of the language itself. Therefore, whenever we say that some function f has an s-line NAND-CIRC program we mean a standard "sugar-free" NAND-CIRC program, where all syntactic sugar has been expanded out. For example, the program of Example 4.3 is a 12-line program for computing the *MAJ* function,

even though it can be written in fewer lines using NAND-CIRC-PROC.

4.1.2 Proof by Python (optional)

We can write a Python program that implements the proof of Theorem 4.1. This is a Python program that takes a NAND-CIRC-PROC program P that includes procedure definitions and uses simple "search and replace" to transform P into a standard (i.e., "sugar-free") NAND-CIRC program P' that computes the same function as P without using any procedures. The idea is simple: if the program P contains a definition of a procedure `Proc` of two arguments `x` and `y`, then whenever we see a line of the form `foo = Proc(bar,blah)`, we can replace this line by:

1. The body of the procedure `Proc` (replacing all occurrences of `x` and `y` with `bar` and `blah` respectively).
2. A line `foo = exp`, where `exp` is the expression following the `return` statement in the definition of the procedure `Proc`.

To make this more robust we add a prefix to the internal variables used by `Proc` to ensure they don't conflict with the variables of P; for simplicity we ignore this issue in the code below though it can be easily added.

The code of the Python function `desugar` below achieves such a transformation.

Fig. 4.2 shows the result of applying `desugar` to the program of Example 4.3 that uses syntactic sugar to compute the Majority function. Specifically, we first apply `desugar` to remove usage of the OR function, then apply it to remove usage of the AND function, and finally apply it a third time to remove usage of the NOT function.

Remark 4.5 — Parsing function definitions (optional). The function `desugar` in Fig. 4.3 assumes that it is given the procedure already split up into its name, arguments, and body. It is not crucial for our purposes to describe precisely how to scan a definition and split it up into these components, but in case you are curious, it can be achieved in Python via the following code:

```
def parse_func(code):
    """Parse a function definition into name,
    ↪  arguments and body"""
    lines = [l.strip() for l in code.split('\n')]
    regexp = r'def\s+([a-zA-Z\_0-9]+)\(([\sa-zA-
    ↪  Z0-9\_,]+)\)\s*:\s*'
```

Figure 4.3: Python code for transforming NAND-CIRC-PROC programs into standard sugar-free NAND-CIRC programs.

```
def desugar(code, func_name, func_args,func_body):
    """
    Replaces all occurences of
       foo = func_name(func_args)
    with
       func_body[x->a,y->b]
       foo = [result returned in func_body]
    """
    # Uses Python regular expressions to simplify the search and replace,
    # see https://docs.python.org/3/library/re.html and Chapter 9 of the book

    # regular expression for capturing a list of variable names separated by commas
    arglist = ",".join([r"([a-zA-Z0-9\_\[\]]+)" for i in range(len(func_args))])
    # regular expression for capturing a statement of the form
    # "variable = func_name(arguments)"
    regexp = fr'([a-zA-Z0-9\_\[\]]+)\s*=\s*{func_name}\({arglist}\)\s*$'
    while True:
        m = re.search(regexp, code, re.MULTILINE)
        if not m: break
        newcode = func_body
        # replace function arguments by the variables from the function invocation
        for i in range(len(func_args)):
            newcode = newcode.replace(func_args[i], m.group(i+2))
        # Splice the new code inside
        newcode = newcode.replace('return', m.group(1) + " = ")
        code = code[:m.start()] + newcode + code[m.end()+1:]
    return code
```

```
        m = re.match(regexp,lines[0])
        return m.group(1), m.group(2).split(','),
         ↪  '\n'.join(lines[1:])
```

4.1.3 Conditional statements

Another sorely missing feature in NAND-CIRC is a conditional statement such as the `if/then` constructs that are found in many programming languages. However, using procedures, we can obtain an ersatz if/then construct. First we can compute the function $IF : \{0,1\}^3 \to \{0,1\}$ such that $IF(a,b,c)$ equals b if $a = 1$ and c if $a = 0$.

Before reading onward, try to see how you could compute the *IF* function using *NAND*'s. Once you do that, see how you can use that to emulate `if/then` types of constructs.

The *IF* function can be implemented from NANDs as follows (see Exercise 4.2):

```
def IF(cond,a,b):
    notcond = NAND(cond,cond)
    temp = NAND(b,notcond)
    temp1 = NAND(a,cond)
    return NAND(temp,temp1)
```

The *IF* function is also known as a *multiplexing* function, since $cond$ can be thought of as a switch that controls whether the output is connected to a or b. Once we have a procedure for computing the *IF* function, we can implement conditionals in NAND. The idea is that we replace code of the form

```
if (condition):  assign blah to variable foo
```

with code of the form

```
foo   = IF(condition, blah, foo)
```

that assigns to `foo` its old value when `condition` equals 0, and assign to `foo` the value of `blah` otherwise. More generally we can replace code of the form

```
if (cond):
    a = ...
    b = ...
    c = ...
```

with code of the form

```
temp_a = ...
temp_b = ...
temp_c = ...
a = IF(cond,temp_a,a)
b = IF(cond,temp_b,b)
c = IF(cond,temp_c,c)
```

Using such transformations, we can prove the following theorem. Once again we omit the (not too insightful) full formal proof, though see Section 4.1.2 for some hints on how to obtain it.

Theorem 4.6 — Conditional statements syntactic sugar. Let NAND-CIRC-IF be the programming language NAND-CIRC augmented with `if/then/else` statements for allowing code to be conditionally executed based on whether a variable is equal to 0 or 1.

Then for every NAND-CIRC-IF program P, there exists a standard (i.e., "sugar-free") NAND-CIRC program P' that computes the same function as P.

4.2 EXTENDED EXAMPLE: ADDITION AND MULTIPLICATION (OPTIONAL)

Using "syntactic sugar", we can write the integer addition function as follows:

```
# Add two n-bit integers
# Use LSB first notation for simplicity
def ADD(A,B):
    Result = [0]*(n+1)
    Carry  = [0]*(n+1)
    Carry[0] = zero(A[0])
    for i in range(n):
        Result[i] = XOR(Carry[i],XOR(A[i],B[i]))
        Carry[i+1] = MAJ(Carry[i],A[i],B[i])
    Result[n] = Carry[n]
    return Result

ADD([1,1,1,0,0],[1,0,0,0,0]);;
# [0, 0, 0, 1, 0, 0]
```

where `zero` is the constant zero function, and `MAJ` and `XOR` correspond to the majority and XOR functions respectively. While we use Python syntax for convenience, in this example n is some *fixed integer* and so for every such n, `ADD` is a *finite* function that takes as input $2n$

bits and outputs $n+1$ bits. In particular for every n we can remove the loop construct `for i in range(n)` by simply repeating the code n times, replacing the value of `i` with $0, 1, 2, \ldots, n-1$. By expanding out all the features, for every value of n we can translate the above program into a standard ("sugar-free") NAND-CIRC program. Fig. 4.4 depicts what we get for $n = 2$.

```
Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(Temp[1],Temp[1])
Temp[3] = NAND(X[0],X[2])
Temp[4] = NAND(X[0],Temp[3])
Temp[5] = NAND(X[2],Temp[3])
Temp[6] = NAND(Temp[4],Temp[5])
Temp[7] = NAND(Temp[2],Temp[6])
Temp[8] = NAND(Temp[2],Temp[7])
Temp[9] = NAND(Temp[6],Temp[7])
Y[0] = NAND(Temp[8],Temp[9])
Temp[11] = NAND(Temp[2],X[0])
Temp[12] = NAND(Temp[11],Temp[11])
Temp[13] = NAND(X[0],X[3])
Temp[14] = NAND(Temp[13],Temp[13])
Temp[15] = NAND(Temp[12],Temp[12])
Temp[16] = NAND(Temp[14],Temp[14])
Temp[17] = NAND(Temp[15],Temp[16])
Temp[18] = NAND(Temp[2],X[2])
Temp[19] = NAND(Temp[18],Temp[18])
Temp[20] = NAND(Temp[17],Temp[17])
Temp[21] = NAND(Temp[19],Temp[19])
Temp[22] = NAND(Temp[20],Temp[21])
Temp[23] = NAND(X[1],X[3])
Temp[24] = NAND(X[1],Temp[23])
Temp[25] = NAND(X[3],Temp[23])
Temp[26] = NAND(Temp[24],Temp[25])
Temp[27] = NAND(Temp[22],Temp[26])
Temp[28] = NAND(Temp[22],Temp[27])
Temp[29] = NAND(Temp[26],Temp[27])
Y[1] = NAND(Temp[28],Temp[29])
Temp[31] = NAND(Temp[22],X[1])
Temp[32] = NAND(Temp[31],Temp[31])
Temp[33] = NAND(X[1],X[3])
Temp[34] = NAND(Temp[33],Temp[33])
Temp[35] = NAND(Temp[32],Temp[32])
Temp[36] = NAND(Temp[34],Temp[34])
Temp[37] = NAND(Temp[35],Temp[36])
Temp[38] = NAND(Temp[22],X[3])
Temp[39] = NAND(Temp[38],Temp[38])
Temp[40] = NAND(Temp[37],Temp[37])
Temp[41] = NAND(Temp[39],Temp[39])
Y[2] = NAND(Temp[40],Temp[41])
```

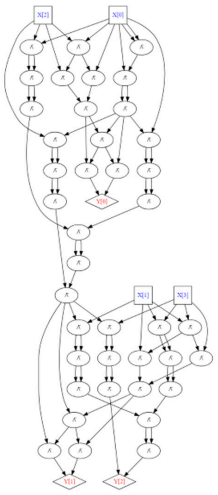


Figure 4.4: The NAND-CIRC program and corresponding NAND circuit for adding two-digit binary numbers that are obtained by "expanding out" all the syntactic sugar. The program/circuit has 43 lines/-gates which is by no means necessary. It is possible to add n bit numbers using $9n$ NAND gates, see Exercise 4.5.

By going through the above program carefully and accounting for the number of gates, we can see that it yields a proof of the following theorem (see also Fig. 4.5):

Theorem 4.7 — Addition using NAND-CIRC programs. For every $n \in \mathbb{N}$, let $ADD_n : \{0,1\}^{2n} \to \{0,1\}^{n+1}$ be the function that, given $x, x' \in \{0,1\}^n$ computes the representation of the sum of the numbers that x and x' represent. Then there is a constant $c \leq 30$ such that for every n there is a NAND-CIRC program of at most cn lines computing ADD_n. [1]

Once we have addition, we can use the grade-school algorithm to obtain multiplication as well, thus obtaining the following theorem:

Theorem 4.8 — Multiplication using NAND-CIRC programs. For every n, let $MULT_n : \{0,1\}^{2n} \to \{0,1\}^{2n}$ be the function that, given $x, x' \in \{0,1\}^n$ computes the representation of the product of the numbers that x and x' represent. Then there is a constant c such that for every n, there is a NAND-CIRC program of at most cn^2 lines that computes the function $MULT_n$.

We omit the proof, though in Exercise 4.7 we ask you to supply a "constructive proof" in the form of a program (in your favorite

[1] The value of c can be improved to 9, see Exercise 4.5.

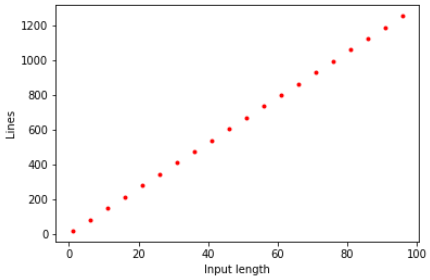


Figure 4.5: The number of lines in our NAND-CIRC program to add two n bit numbers, as a function of n, for n's between 1 and 100. This is not the most efficient program for this task, but the important point is that it has the form $O(n)$.

programming language) that on input a number n, outputs the code of a NAND-CIRC program of at most $1000n^2$ lines that computes the $MULT_n$ function. In fact, we can use Karatsuba's algorithm to show that there is a NAND-CIRC program of $O(n^{\log_2 3})$ lines to compute $MULT_n$ (and can get even further asymptotic improvements using better algorithms).

4.3 THE LOOKUP FUNCTION

The *LOOKUP* function will play an important role in this chapter and later. It is defined as follows:

Definition 4.9 — Lookup function. For every k, the *lookup* function of order k, $LOOKUP_k : \{0,1\}^{2^k+k} \rightarrow \{0,1\}$ is defined as follows: For every $x \in \{0,1\}^{2^k}$ and $i \in \{0,1\}^k$,

$$LOOKUP_k(x,i) = x_i$$

where x_i denotes the ith entry of x, using the binary representation to identify i with a number in $\{0,\ldots,2^k-1\}$.

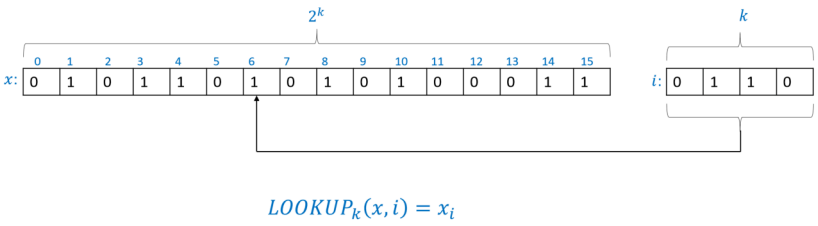


Figure 4.6: The $LOOKUP_k$ function takes an input in $\{0,1\}^{2^k+k}$, which we denote by x, i (with $x \in \{0,1\}^{2^k}$ and $i \in \{0,1\}^k$). The output is x_i: the i-th coordinate of x, where we identify i as a number in $[k]$ using the binary representation. In the above example $x \in \{0,1\}^{16}$ and $i \in \{0,1\}^4$. Since $i = 0110$ is the binary representation of the number 6, the output of $LOOKUP_4(x,i)$ in this case is $x_6 = 1$.

See Fig. 4.6 for an illustration of the LOOKUP function. It turns out that for every k, we can compute $LOOKUP_k$ using a NAND-CIRC program:

Theorem 4.10 — Lookup function. For every $k > 0$, there is a NAND-CIRC program that computes the function $LOOKUP_k : \{0,1\}^{2^k+k} \rightarrow \{0,1\}$. Moreover, the number of lines in this program is at most $4 \cdot 2^k$.

An immediate corollary of Theorem 4.10 is that for every $k > 0$, $LOOKUP_k$ can be computed by a Boolean circuit (with AND, OR and NOT gates) of at most $8 \cdot 2^k$ gates.

4.3.1 Constructing a NAND-CIRC program for *LOOKUP*

We prove Theorem 4.10 by induction. For the case $k = 1$, $LOOKUP_1$ maps $(x_0, x_1, i) \in \{0,1\}^3$ to x_i. In other words, if $i = 0$ then it outputs

x_0 and otherwise it outputs x_1, which (up to reordering variables) is the same as the *IF* function presented in Section 4.1.3, which can be computed by a 4-line NAND-CIRC program.

As a warm-up for the case of general k, let us consider the case of $k = 2$. Given input $x = (x_0, x_1, x_2, x_3)$ for $LOOKUP_2$ and an index $i = (i_0, i_1)$, if the most significant bit i_0 of the index is 0 then $LOOKUP_2(x, i)$ will equal x_0 if $i_1 = 0$ and equal x_1 if $i_1 = 1$. Similarly, if the most significant bit i_0 is 1 then $LOOKUP_2(x, i)$ will equal x_2 if $i_1 = 0$ and will equal x_3 if $i_1 = 1$. Another way to say this is that we can write $LOOKUP_2$ as follows:

```
def LOOKUP2(X[0],X[1],X[2],X[3],i[0],i[1]):
    if i[0]==1:
        return LOOKUP1(X[2],X[3],i[1])
    else:
        return LOOKUP1(X[0],X[1],i[1])
```

or in other words,

```
def LOOKUP2(X[0],X[1],X[2],X[3],i[0],i[1]):
    a = LOOKUP1(X[2],X[3],i[1])
    b = LOOKUP1(X[0],X[1],i[1])
    return IF( i[0],a,b)
```

More generally, as shown in the following lemma, we can compute $LOOKUP_k$ using two invocations of $LOOKUP_{k-1}$ and one invocation of *IF*:

Lemma 4.11 — Lookup recursion. For every $k \geq 2$, $LOOKUP_k(x_0, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1})$ is equal to

$$IF\left(i_0, LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_1, \ldots, i_{k-1}), LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_1, \ldots, i_{k-1})\right)$$

Proof. If the most significant bit i_0 of i is zero, then the index i is in $\{0, \ldots, 2^{k-1} - 1\}$ and hence we can perform the lookup on the "first half" of x and the result of $LOOKUP_k(x, i)$ will be the same as $a = LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_1, \ldots, i_{k-1})$. On the other hand, if this most significant bit i_0 is equal to 1, then the index is in $\{2^{k-1}, \ldots, 2^k - 1\}$, in which case the result of $LOOKUP_k(x, i)$ is the same as $b = LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_1, \ldots, i_{k-1})$. Thus we can compute $LOOKUP_k(x, i)$ by first computing a and b and then outputting $IF(i_{k-1}, a, b)$.

■

Proof of Theorem 4.10 from Lemma 4.11. Now that we have Lemma 4.11, we can complete the proof of Theorem 4.10. We will prove by induction on k that there is a NAND-CIRC program of at most $4 \cdot (2^k - 1)$

lines for $LOOKUP_k$. For $k = 1$ this follows by the four line program for *IF* we've seen before. For $k > 1$, we use the following pseudocode:

```
a = LOOKUP_(k-1)(X[0],...,X[2^(k-1)-1],i[1],...,i[k-1])
b = LOOKUP_(k-1)(X[2^(k-1)],...,Z[2^(k-1)],i[1],...,i[k-
 ↪  1])
return IF(i[0],b,a)
```

If we let $L(k)$ be the number of lines required for $LOOKUP_k$, then the above pseudo-code shows that

$$L(k) \leq 2L(k-1) + 4 \,. \tag{4.1}$$

Since under our induction hypothesis $L(k-1) \leq 4(2^{k-1} - 1)$, we get that $L(k) \leq 2 \cdot 4(2^{k-1} - 1) + 4 = 4(2^k - 1)$ which is what we wanted to prove. See Fig. 4.7 for a plot of the actual number of lines in our implementation of $LOOKUP_k$.

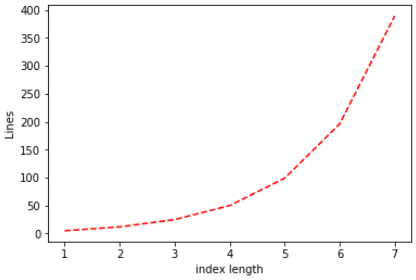


Figure 4.7: The number of lines in our implementation of the LOOKUP_k function as a function of k (i.e., the length of the index). The number of lines in our implementation is roughly $3 \cdot 2^k$.

4.4 COMPUTING *EVERY* FUNCTION

At this point we know the following facts about NAND-CIRC programs (and so equivalently about Boolean circuits and our other equivalent models):

1. They can compute at least some non-trivial functions.
2. Coming up with NAND-CIRC programs for various functions is a very tedious task.

Thus I would not blame the reader if they were not particularly looking forward to a long sequence of examples of functions that can be computed by NAND-CIRC programs. However, it turns out we are not going to need this, as we can show in one fell swoop that NAND-CIRC programs can compute *every* finite function:

Theorem 4.12 — Universality of NAND. There exists some constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0,1\}^n \to \{0,1\}^m$, there is a NAND-CIRC program with at most $c \cdot m 2^n$ lines that computes the function f.

By Theorem 3.19, the models of NAND circuits, NAND-CIRC programs, AON-CIRC programs, and Boolean circuits, are all equivalent to one another, and hence Theorem 4.12 holds for all these models. In particular, the following theorem is equivalent to Theorem 4.12:

Theorem 4.13 — Universality of Boolean circuits. There exists some constant $c > 0$ such that for every $n, m > 0$ and function

> $f : \{0,1\}^n \to \{0,1\}^m$, there is a Boolean circuit with at most $c \cdot m2^n$ gates that computes the function f.

Big Idea 5 *Every* finite function can be computed by a large enough Boolean circuit.

Improved bounds. Though it will not be of great importance to us, it is possible to improve on the proof of Theorem 4.12 and shave an extra factor of n, as well as optimize the constant c, and so prove that for every $\epsilon > 0$, $m \in \mathbb{N}$ and sufficiently large n, if $f : \{0,1\}^n \to \{0,1\}^m$ then f can be computed by a NAND circuit of at most $(1+\epsilon)\frac{m \cdot 2^n}{n}$ gates. The proof of this result is beyond the scope of this book, but we do discuss how to obtain a bound of the form $O(\frac{m \cdot 2^n}{n})$ in Section 4.4.2; see also the biographical notes.

4.4.1 Proof of NAND's Universality

To prove Theorem 4.12, we need to give a NAND circuit, or equivalently a NAND-CIRC program, for *every* possible function. We will restrict our attention to the case of Boolean functions (i.e., $m = 1$). Exercise 4.9 asks you to extend the proof for all values of m. A function $F : \{0,1\}^n \to \{0,1\}$ can be specified by a table of its values for each one of the 2^n inputs. For example, the table below describes one particular function $G : \{0,1\}^4 \to \{0,1\}$:[2]

[2] In case you are curious, this is the function on input $i \in \{0,1\}^4$ (which we interpret as a number in $[16]$), that outputs the i-th digit of π in the binary basis.

Table 4.1: An example of a function $G : \{0,1\}^4 \to \{0,1\}$.

Input (x)	Output ($G(x)$)
0000	1
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	1
1000	0
1001	0
1010	0
1011	0
1100	1
1101	1
1110	1
1111	1

For every $x \in \{0,1\}^4$, $G(x) = LOOKUP_4(1100100100001111, x)$, and so the following is NAND-CIRC "pseudocode" to compute G using syntactic sugar for the `LOOKUP_4` procedure.

```
G0000 = 1
G1000 = 1
G0100 = 0
...
G0111 = 1
G1111 = 1
Y[0] = LOOKUP_4(G0000,G1000,...,G1111,
                X[0],X[1],X[2],X[3])
```

We can translate this pseudocode into an actual NAND-CIRC program by adding three lines to define variables `zero` and `one` that are initialized to 0 and 1 respectively, and then replacing a statement such as `Gxxx = 0` with `Gxxx = NAND(one,one)` and a statement such as `Gxxx = 1` with `Gxxx = NAND(zero,zero)`. The call to `LOOKUP_4` will be replaced by the NAND-CIRC program that computes $LOOKUP_4$, plugging in the appropriate inputs.

There was nothing about the above reasoning that was particular to the function G above. Given *every* function $F : \{0,1\}^n \to \{0,1\}$, we can write a NAND-CIRC program that does the following:

1. Initialize 2^n variables of the form `F00...0` till `F11...1` so that for every $z \in \{0,1\}^n$, the variable corresponding to z is assigned the value $F(z)$.
2. Compute $LOOKUP_n$ on the 2^n variables initialized in the previous step, with the index variable being the input variables `X[0]`,...,`X[`$n-1$`]`. That is, just like in the pseudocode for `G` above, we use `Y[0] = LOOKUP(F00..00,...,F11..1,X[0],..,X[`$n-1$`])`

The total number of lines in the resulting program is $3 + 2^n$ lines for initializing the variables plus the $4 \cdot 2^n$ lines that we pay for computing $LOOKUP_n$. This completes the proof of Theorem 4.12.

R

Remark 4.14 — Result in perspective. While Theorem 4.12 seems striking at first, in retrospect, it is perhaps not that surprising that every finite function can be computed with a NAND-CIRC program. After all, a finite function $F : \{0,1\}^n \to \{0,1\}^m$ can be represented by simply the list of its outputs for each one of the 2^n input values. So it makes sense that we could write a NAND-CIRC program of similar size to compute it. What is more interesting is that *some* functions, such as addition and multiplication, have

a much more efficient representation: one that only requires $O(n^2)$ or even fewer lines.

4.4.2 Improving by a factor of n (optional)

By being a little more careful, we can improve the bound of Theorem 4.12 and show that every function $F : \{0,1\}^n \to \{0,1\}^m$ can be computed by a NAND-CIRC program of at most $O(m2^n/n)$ lines. In other words, we can prove the following improved version:

Theorem 4.15 — Universality of NAND circuits, improved bound. There exists a constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0,1\}^n \to \{0,1\}^m$, there is a NAND-CIRC program with at most $c \cdot m2^n/n$ lines that computes the function f. [3]

[3] The constant c in this theorem is at most 10 and in fact can be arbitrarily close to 1, see Section 4.8.

Proof. As before, it is enough to prove the case that $m = 1$. Hence we let $f : \{0,1\}^n \to \{0,1\}$, and our goal is to prove that there exists a NAND-CIRC program of $O(2^n/n)$ lines (or equivalently a Boolean circuit of $O(2^n/n)$ gates) that computes f.

We let $k = \log(n - 2\log n)$ (the reasoning behind this choice will become clear later on). We define the function $g : \{0,1\}^k \to \{0,1\}^{2^{n-k}}$ as follows:

$$g(a) = f(a0^{n-k})f(a0^{n-k-1}1)\cdots f(a1^{n-k}) \ .$$

In other words, if we use the usual binary representation to identify the numbers $\{0,\ldots,2^{n-k}-1\}$ with the strings $\{0,1\}^{n-k}$, then for every $a \in \{0,1\}^k$ and $b \in \{0,1\}^{n-k}$

$$g(a)_b = f(ab) \ . \tag{4.2}$$

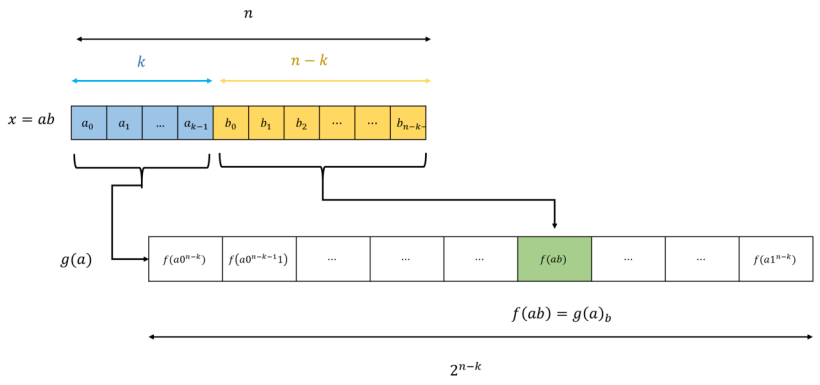


Figure 4.8: We can compute $f : \{0,1\}^n \to \{0,1\}$ on input $x = ab$ where $a \in \{0,1\}^k$ and $b \in \{0,1\}^{n-k}$ by first computing the 2^{n-k} long string $g(a)$ that corresponds to all f's values on inputs that begin with a, and then outputting the b-th coordinate of this string.

(4.2) means that for every $x \in \{0,1\}^n$, if we write $x = ab$ with $a \in \{0,1\}^k$ and $b \in \{0,1\}^{n-k}$ then we can compute $f(x)$ by first

computing the string $T = g(a)$ of length 2^{n-k}, and then computing $LOOKUP_{n-k}(T\ ,\ b)$ to retrieve the element of T at the position corresponding to b (see Fig. 4.8). The cost to compute the $LOOKUP_{n-k}$ is $O(2^{n-k})$ lines/gates and the cost in NAND-CIRC lines (or Boolean gates) to compute f is at most

$$cost(g) + O(2^{n-k}) \ , \tag{4.3}$$

where $cost(g)$ is the number of operations (i.e., lines of NAND-CIRC programs or gates in a circuit) needed to compute g.

To complete the proof we need to give a bound on $cost(g)$. Since g is a function mapping $\{0,1\}^k$ to $\{0,1\}^{2^{n-k}}$, we can also think of it as a collection of 2^{n-k} functions $g_0, \ldots, g_{2^{n-k}-1} : \{0,1\}^k \to \{0,1\}$, where $g_i(x) = g(a)_i$ for every $a \in \{0,1\}^k$ and $i \in [2^{n-k}]$. (That is, $g_i(a)$ is the i-th bit of $g(a)$.) Naively, we could use Theorem 4.12 to compute each g_i in $O(2^k)$ lines, but then the total cost is $O(2^{n-k} \cdot 2^k) = O(2^n)$ which does not save us anything. However, the crucial observation is that there are only 2^{2^k} *distinct functions* mapping $\{0,1\}^k$ to $\{0,1\}$. For example, if g_{17} is an identical function to g_{67} that means that if we already computed $g_{17}(a)$ then we can compute $g_{67}(a)$ using only a constant number of operations: simply copy the same value! In general, if you have a collection of N functions $g_0, \ldots, g_{N-1}$ mapping $\{0,1\}^k$ to $\{0,1\}$, of which at most S are distinct then for every value $a \in \{0,1\}^k$ we can compute the N values $g_0(a), \ldots, g_{N-1}(a)$ using at most $O(S \cdot 2^k + N)$ operations (see Fig. 4.9).

In our case, because there are at most 2^{2^k} distinct functions mapping $\{0,1\}^k$ to $\{0,1\}$, we can compute the function g (and hence by (4.2) also f) using at most

$$O(2^{2^k} \cdot 2^k + 2^{n-k}) \tag{4.4}$$

operations. Now all that is left is to plug into (4.4) our choice of $k = \log(n - 2\log n)$. By definition, $2^k = n - 2\log n$, which means that (4.4) can be bounded

$$O\left(2^{n-2\log n} \cdot (n - 2\log n) + 2^{n-\log(n-2\log n)}\right) \leq$$

$$O\left(\tfrac{2^n}{n^2} \cdot n + \tfrac{2^n}{n-2\log n}\right) \leq O\left(\tfrac{2^n}{n} + \tfrac{2^n}{0.5n}\right) = O\left(\tfrac{2^n}{n}\right)$$

which is what we wanted to prove. (We used above the fact that $n - 2\log n \geq 0.5\log n$ for sufficiently large n.)

■

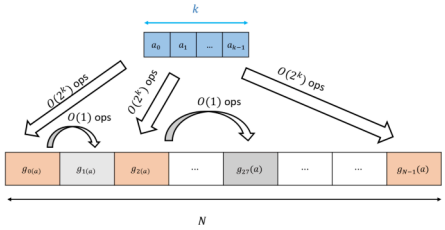


Figure 4.9: If $g_0, \ldots, g_{N-1}$ is a collection of functions each mapping $\{0,1\}^k$ to $\{0,1\}$ such that at most S of them are distinct then for every $a \in \{0,1\}^k$, we can compute all the values $g_0(a), \ldots, g_{N-1}(a)$ using at most $O(S \cdot 2^k + N)$ operations by first computing the distinct functions and then copying the resulting values.

Using the connection between NAND-CIRC programs and Boolean circuits, an immediate corollary of Theorem 4.15 is the following improvement to Theorem 4.13:

Theorem 4.16 — Universality of Boolean circuits, improved bound. There exists some constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0,1\}^n \to \{0,1\}^m$, there is a Boolean circuit with at most $c \cdot m2^n/n$ gates that computes the function f.

4.5 COMPUTING EVERY FUNCTION: AN ALTERNATIVE PROOF

Theorem 4.13 is a fundamental result in the theory (and practice!) of computation. In this section, we present an alternative proof of this basic fact that Boolean circuits can compute every finite function. This alternative proof gives a somewhat worse quantitative bound on the number of gates but it has the advantage of being simpler, working directly with circuits and avoiding the usage of all the syntactic sugar machinery. (However, that machinery is useful in its own right, and will find other applications later on.)

Theorem 4.17 — Universality of Boolean circuits (alternative phrasing). There exists some constant $c > 0$ such that for every $n, m > 0$ and function $f : \{0,1\}^n \to \{0,1\}^m$, there is a Boolean circuit with at most $c \cdot m \cdot n2^n$ gates that computes the function f.

Proof Idea:

The idea of the proof is illustrated in Fig. 4.10. As before, it is enough to focus on the case that $m = 1$ (the function f has a single output), since we can always extend this to the case of $m > 1$ by looking at the composition of m circuits each computing a different output bit of the function f. We start by showing that for every $\alpha \in \{0,1\}^n$, there is an $O(n)$-sized circuit that computes the function $\delta_\alpha : \{0,1\}^n \to \{0,1\}$ defined as follows: $\delta_\alpha(x) = 1$ iff $x = \alpha$ (that is, δ_α outputs 0 on all inputs except the input α). We can then write any function $f : \{0,1\}^n \to \{0,1\}$ as the OR of at most 2^n functions δ_α for the α's on which $f(\alpha) = 1$.

⋆

Proof of Theorem 4.17. We prove the theorem for the case $m = 1$. The result can be extended for $m > 1$ as before (see also Exercise 4.9). Let $f : \{0,1\}^n \to \{0,1\}$. We will prove that there is an $O(n \cdot 2^n)$-sized Boolean circuit to compute f in the following steps:

1. We show that for every $\alpha \in \{0,1\}^n$, there is an $O(n)$-sized circuit that computes the function $\delta_\alpha : \{0,1\}^n \to \{0,1\}$, where $\delta_\alpha(x) = 1$ iff $x = \alpha$.
2. We then show that this implies the existence of an $O(n \cdot 2^n)$-sized circuit that computes f, by writing $f(x)$ as the OR of $\delta_\alpha(x)$ for all

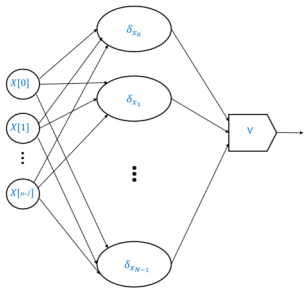


Figure 4.10: Given a function $f : \{0,1\}^n \to \{0,1\}$, we let $\{x_0, x_1, \ldots, x_{N-1}\} \subseteq \{0,1\}^n$ be the set of inputs such that $f(x_i) = 1$, and note that $N \leq 2^n$. We can express f as the OR of δ_{x_i} for $i \in [N]$ where the function $\delta_\alpha : \{0,1\}^n \to \{0,1\}$ (for $\alpha \in \{0,1\}^n$) is defined as follows: $\delta_\alpha(x) = 1$ iff $x = \alpha$. We can compute the OR of N values using N two-input OR gates. Therefore if we have a circuit of size $O(n)$ to compute δ_α for every $\alpha \in \{0,1\}^n$, we can compute f using a circuit of size $O(n \cdot N) = O(n \cdot 2^n)$.

$\alpha \in \{0,1\}^n$ such that $f(\alpha) = 1$. (If f is the constant zero function and hence there is no such α, then we can use the circuit $f(x) = x_0 \wedge \overline{x}_0$.)

We start with Step 1:

CLAIM: For $\alpha \in \{0,1\}^n$, define $\delta_\alpha : \{0,1\}^n$ as follows:

$$\delta_\alpha(x) = \begin{cases} 1 & x = \alpha \\ 0 & \text{otherwise} \end{cases} .$$

then there is a Boolean circuit using at most $2n$ gates that computes δ_α.

PROOF OF CLAIM: The proof is illustrated in Fig. 4.11. As an example, consider the function $\delta_{011} : \{0,1\}^3 \to \{0,1\}$. This function outputs 1 on x if and only if $x_0 = 0$, $x_1 = 1$ and $x_2 = 1$, and so we can write $\delta_{011}(x) = \overline{x_0} \wedge x_1 \wedge x_2$, which translates into a Boolean circuit with one NOT gate and two AND gates. More generally, for every $\alpha \in \{0,1\}^n$, we can express $\delta_\alpha(x)$ as $(x_0 = \alpha_0) \wedge (x_1 = \alpha_1) \wedge \cdots \wedge (x_{n-1} = \alpha_{n-1})$, where if $\alpha_i = 0$ we replace $x_i = \alpha_i$ with $\overline{x_i}$ and if $\alpha_i = 1$ we replace $x_i = \alpha_i$ by simply x_i. This yields a circuit that computes δ_α using n AND gates and at most n NOT gates, so a total of at most $2n$ gates.

Now for every function $f : \{0,1\}^n \to \{0,1\}$, we can write

$$f(x) = \delta_{x_0}(x) \vee \delta_{x_1}(x) \vee \cdots \vee \delta_{x_{N-1}}(x) \tag{4.5}$$

where $S = \{x_0, \ldots, x_{N-1}\}$ is the set of inputs on which f outputs 1. (To see this, you can verify that the right-hand side of (4.5) evaluates to 1 on $x \in \{0,1\}^n$ if and only if x is in the set S.)

Therefore we can compute f using a Boolean circuit of at most $2n$ gates for each of the N functions δ_{x_i} and combine that with at most N OR gates, thus obtaining a circuit of at most $2n \cdot N + N$ gates. Since $S \subseteq \{0,1\}^n$, its size N is at most 2^n and hence the total number of gates in this circuit is $O(n \cdot 2^n)$. ■

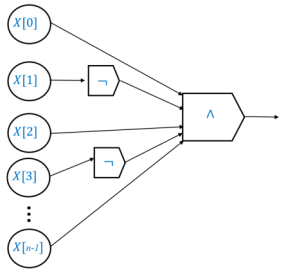


Figure 4.11: For every string $\alpha \in \{0,1\}^n$, there is a Boolean circuit of $O(n)$ gates to compute the function $\delta_\alpha : \{0,1\}^n \to \{0,1\}$ such that $\delta_\alpha(x) = 1$ if and only if $x = \alpha$. The circuit is very simple. Given input $x_0, \ldots, x_{n-1}$ we compute the AND of $z_0, \ldots, z_{n-1}$ where $z_i = x_i$ if $\alpha_i = 1$ and $z_i = NOT(x_i)$ if $\alpha_i = 0$. While formally Boolean circuits only have a gate for computing the AND of two inputs, we can implement an AND of n inputs by composing n two-input ANDs.

4.6 THE CLASS $SIZE_{n,m}(s)$

We have seen that *every* function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a circuit of size $O(m \cdot 2^n)$, and *some* functions (such as addition and multiplication) can be computed by much smaller circuits. We define $SIZE_{n,m}(s)$ to be the set of functions mapping n bits to m bits that can be computed by NAND circuits of at most s gates (or equivalently, by NAND-CIRC programs of at most s lines). Formally, the definition is as follows:

Definition 4.18 — Size class of functions. For all natural numbers n, m, s, let $SIZE_{n,m}(s)$ denote the set of all functions $f : \{0,1\}^n \to \{0,1\}^m$ such that there exists a NAND circuit of at most s gates computing f. We denote by $SIZE_n(s)$ the set $SIZE_{n,1}(s)$. For every integer $s \geq 1$, we let $SIZE(s) = \cup_{n,m} SIZE_{n,m}(s)$ be the set of all functions f for which there exists a NAND circuit of at most s gates that compute f.

Fig. 4.12 depicts the set $SIZE_{n,1}(s)$. Note that $SIZE_{n,m}(s)$ is a set of *functions*, not of *programs!* Asking if a program or a circuit is a member of $SIZE_{n,m}(s)$ is a *category error* as in the sense of Fig. 4.13. As we discussed in Section 3.7.2 (and Section 2.6.1), the distinction between *programs* and *functions* is absolutely crucial. You should always remember that while a program *computes* a function, it is not *equal* to a function. In particular, as we've seen, there can be more than one program to compute the same function.

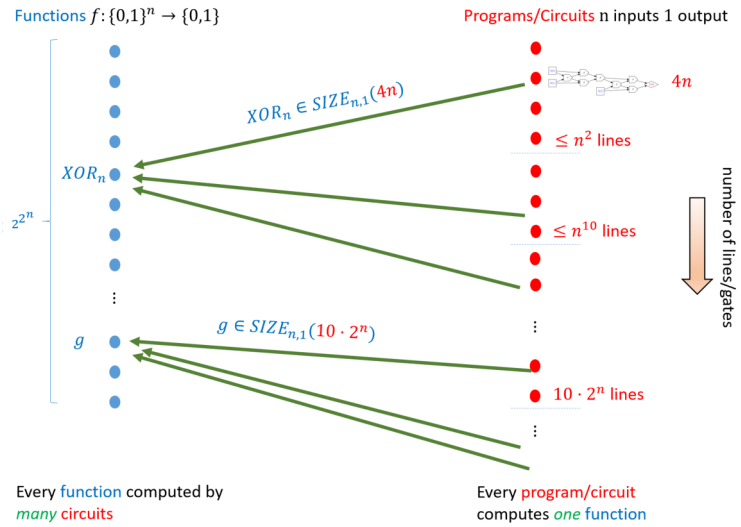


Figure 4.12: There are 2^{2^n} functions mapping $\{0,1\}^n$ to $\{0,1\}$, and an infinite number of circuits with n bit inputs and a single bit of output. Every circuit computes one function, but every function can be computed by many circuits. We say that $f \in SIZE_{n,1}(s)$ if the smallest circuit that computes f has s or fewer gates. For example $XOR_n \in SIZE_{n,1}(4n)$. Theorem 4.12 shows that *every* function g is computable by some circuit of at most $c \cdot 2^n/n$ gates, and hence $SIZE_{n,1}(c \cdot 2^n/n)$ corresponds to the set of *all* functions from $\{0,1\}^n$ to $\{0,1\}$.

While we defined $SIZE_n(s)$ with respect to NAND gates, we would get essentially the same class if we defined it with respect to AND/OR/NOT gates:

Lemma 4.19 Let $SIZE^{AON}_{n,m}(s)$ denote the set of all functions $f : \{0,1\}^n \to \{0,1\}^m$ that can be computed by an AND/OR/NOT Boolean circuit of at most s gates. Then,

$$SIZE_{n,m}(s/2) \subseteq SIZE^{AON}_{n,m}(s) \subseteq SIZE_{n,m}(3s)$$

Proof. If f can be computed by a NAND circuit of at most $s/2$ gates, then by replacing each NAND with the two gates NOT and AND, we can obtain an AND/OR/NOT Boolean circuit of at most s gates that

computes f. On the other hand, if f can be computed by a Boolean AND/OR/NOT circuit of at most s gates, then by Theorem 3.12 it can be computed by a NAND circuit of at most $3s$ gates.

■

The results we have seen in this chapter can be phrased as showing that $ADD_n \in SIZE_{2n,n+1}(100n)$ and $MULT_n \in SIZE_{2n,2n}(10000n^{\log_2 3})$. Theorem 4.12 shows that for some constant c, $SIZE_{n,m}(cm2^n)$ is equal to the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$.

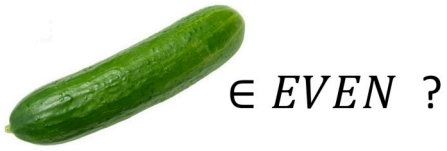


Figure 4.13: A "category error" is a question such as "is a cucumber even or odd?" which does not even make sense. In this book one type of category error you should watch out for is confusing *functions* and *programs* (i.e., confusing *specifications* and *implementations*). If C is a circuit or program, then asking if $C \in SIZE_{n,1}(s)$ is a category error, since $SIZE_{n,1}(s)$ is a set of *functions* and not programs or circuits.

R

Remark 4.20 — Finite vs infinite functions. Unlike programming languages such as *Python*, *C* or *JavaScript*, the NAND-CIRC and AON-CIRC programming language do not have *arrays*. A NAND-CIRC program P has some fixed number n and m of inputs and output variable. Hence, for example, there is no single NAND-CIRC program that can compute the increment function $INC : \{0,1\}^* \to \{0,1\}^*$ that maps a string x (which we identify with a number via the binary representation) to the string that represents $x+1$. Rather for every $n > 0$, there is a NAND-CIRC program P_n that computes the restriction INC_n of the function INC to inputs of length n. Since it can be shown that for every $n > 0$ such a program P_n exists of length at most $10n$, $INC_n \in SIZE_{n,n+1}(10n)$ for every $n > 0$.

For the time being, our focus will be on *finite* functions, but we will discuss how to extend the definition of size complexity to functions with unbounded input lengths later on in Section 13.6.

Solved Exercise 4.1 — $SIZE$ closed under complement.. In this exercise we prove a certain "closure property" of the class $SIZE_n(s)$. That is, we show that if f is in this class then (up to some small additive term) so is the complement of f, which is the function $g(x) = 1 - f(x)$.

Prove that there is a constant c such that for every $f : \{0,1\}^n \to \{0,1\}$ and $s \in \mathbb{N}$, if $f \in SIZE_n(s)$ then $1 - f \in SIZE_n(s+c)$.

■

Solution:

If $f \in SIZE_n(s)$ then there is an s-line NAND-CIRC program P that computes f. We can rename the variable `Y[0]` in P to a variable `temp` and add the line

```
Y[0] = NAND(temp,temp)
```

at the very end to obtain a program P' that computes $1 - f$.

Chapter Recap

- We can define the notion of computing a function via a simplified "programming language", where computing a function F in T steps would correspond to having a T-line NAND-CIRC program that computes F.
- While the NAND-CIRC programming only has one operation, other operations such as functions and conditional execution can be implemented using it.
- Every function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a circuit of at most $O(m2^n)$ gates (and in fact at most $O(m2^n/n)$ gates).
- Sometimes (or maybe always?) we can translate an *efficient* algorithm to compute f into a circuit that computes f with a number of gates comparable to the number of steps in this algorithm.

4.7 EXERCISES

Exercise 4.1 — Pairing. This exercise asks you to give a one-to-one map from $\mathbb{N}^2$ to $\mathbb{N}$. This can be useful to implement two-dimensional arrays as "syntactic sugar" in programming languages that only have one-dimensional arrays.

1. Prove that the map $F(x,y) = 2^x 3^y$ is a one-to-one map from $\mathbb{N}^2$ to $\mathbb{N}$.

2. Show that there is a one-to-one map $F : \mathbb{N}^2 \to \mathbb{N}$ such that for every x, y, $F(x,y) \leq 100 \cdot \max\{x,y\}^2 + 100$.

3. For every k, show that there is a one-to-one map $F : \mathbb{N}^k \to \mathbb{N}$ such that for every $x_0, \ldots, x_{k-1} \in \mathbb{N}$, $F(x_0, \ldots, x_{k-1}) \leq 100 \cdot (x_0 + x_1 + \ldots + x_{k-1} + 100k)^k$.

Exercise 4.2 — Computing MUX. Prove that the NAND-CIRC program below computes the function MUX (or $LOOKUP_1$) where $MUX(a,b,c)$ equals a if $c = 0$ and equals b if $c = 1$:

```
t = NAND(X[2],X[2])
u = NAND(X[0],t)
v = NAND(X[1],X[2])
Y[0] = NAND(u,v)
```

Exercise 4.3 — At least two / Majority. Give a NAND-CIRC program of at most 6 lines to compute the function $MAJ : \{0,1\}^3 \to \{0,1\}$ where $MAJ(a,b,c) = 1$ iff $a + b + c \geq 2$.

■

Exercise 4.4 — Conditional statements. In this exercise we will explore Theorem 4.6: transforming NAND-CIRC-IF programs that use code such as `if .. then .. else ..` to standard NAND-CIRC programs.

1. Give a "proof by code" of Theorem 4.6: a program in a programming language of your choice that transforms a NAND-CIRC-IF program P into a "sugar-free" NAND-CIRC program P' that computes the same function. See footnote for hint.[4]

2. Prove the following statement, which is the heart of Theorem 4.6: suppose that there exists an s-line NAND-CIRC program to compute $f : \{0,1\}^n \to \{0,1\}$ and an s'-line NAND-CIRC program to compute $g : \{0,1\}^n \to \{0,1\}$. Prove that there exist a NAND-CIRC program of at most $s + s' + 10$ lines to compute the function $h : \{0,1\}^{n+1} \to \{0,1\}$ where $h(x_0, \ldots, x_{n-1}, x_n)$ equals $f(x_0, \ldots, x_{n-1})$ if $x_n = 0$ and equals $g(x_0, \ldots, x_{n-1})$ otherwise. (All programs in this item are standard "sugar-free" NAND-CIRC programs.)

[4] You can start by transforming P into a NAND-CIRC-PROC program that uses procedure statements, and then use the code of Fig. 4.3 to transform the latter into a "sugar-free" NAND-CIRC program.

■

Exercise 4.5 — Half and full adders. 1. A *half adder* is the function $HA : \{0,1\}^2 :\to \{0,1\}^2$ that corresponds to adding two binary bits. That is, for every $a, b \in \{0,1\}$, $HA(a,b) = (e,f)$ where $2e + f = a + b$. Prove that there is a NAND circuit of at most five NAND gates that computes HA.

2. A *full adder* is the function $FA : \{0,1\}^3 \to \{0,1\}^2$ that takes in two bits and a "carry" bit and outputs their sum. That is, for every $a, b, c \in \{0,1\}$, $FA(a,b,c) = (e,f)$ such that $2e + f = a + b + c$. Prove that there is a NAND circuit of at most nine NAND gates that computes FA.

3. Prove that if there is a NAND circuit of c gates that computes FA, then there is a circuit of cn gates that computes ADD_n where (as in Theorem 4.7) $ADD_n : \{0,1\}^{2n} \to \{0,1\}^{n+1}$ is the function that outputs the addition of two input n-bit numbers. See footnote for hint.[5]

4. Show that for every n there is a NAND-CIRC program to compute ADD_n with at most $9n$ lines.

[5] Use a "cascade" of adding the bits one after the other, starting with the least significant digit, just like in the elementary-school algorithm.

■

Exercise 4.6 — Addition. Write a program using your favorite programming language that on input of an integer n, outputs a NAND-CIRC program that computes ADD_n. Can you ensure that the program it outputs for ADD_n has fewer than $10n$ lines?

■

Exercise 4.7 — Multiplication. Write a program using your favorite programming language that on input of an integer n, outputs a NAND-CIRC program that computes $MULT_n$. Can you ensure that the program it outputs for $MULT_n$ has fewer than $1000 \cdot n^2$ lines?

■

Exercise 4.8 — Efficient multiplication (challenge). Write a program using your favorite programming language that on input of an integer n, outputs a NAND-CIRC program that computes $MULT_n$ and has at most $10000n^{1.9}$ lines.[6] What is the smallest number of lines you can use to multiply two 2048 bit numbers?

[6] **Hint:** Use Karatsuba's algorithm.

■

Exercise 4.9 — Multibit function. In the text Theorem 4.12 is only proven for the case $m = 1$. In this exercise you will extend the proof for every m.

Prove that

1. If there is an s-line NAND-CIRC program to compute $f : \{0,1\}^n \rightarrow \{0,1\}$ and an s'-line NAND-CIRC program to compute $f' : \{0,1\}^n \rightarrow \{0,1\}$ then there is an $s + s'$-line program to compute the function $g : \{0,1\}^n \rightarrow \{0,1\}^2$ such that $g(x) = (f(x), f'(x))$.

2. For every function $f : \{0,1\}^n \rightarrow \{0,1\}^m$, there is a NAND-CIRC program of at most $10m \cdot 2^n$ lines that computes f. (You can use the $m = 1$ case of Theorem 4.12, as well as Item 1.)

■

Exercise 4.10 — Simplifying using syntactic sugar. Let P be the following NAND-CIRC program:

```
Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[1],X[1])
Temp[2] = NAND(Temp[0],Temp[1])
Temp[3] = NAND(X[2],X[2])
Temp[4] = NAND(X[3],X[3])
Temp[5] = NAND(Temp[3],Temp[4])
Temp[6] = NAND(Temp[2],Temp[2])
Temp[7] = NAND(Temp[5],Temp[5])
Y[0] = NAND(Temp[6],Temp[7])
```

1. Write a program P' with at most three lines of code that uses both `NAND` as well as the syntactic sugar `OR` that computes the same function as P.

2. Draw a circuit that computes the same function as P and uses only *AND* and *NOT* gates.

■

In the following exercises you are asked to compare the *power* of pairs of programming languages. By "comparing the power" of two programming languages X and Y we mean determining the relation between the set of functions that are computable using programs in X and Y respectively. That is, to answer such a question you need to do both of the following:

1. Either prove that for every program P in X there is a program P' in Y that computes the same function as P, *or* give an example for a function that is computable by an X-program but not computable by a Y-program.

   and

2. Either prove that for every program P in Y there is a program P' in X that computes the same function as P, *or* give an example for a function that is computable by a Y-program but not computable by an X-program.

When you give an example as above of a function that is computable in one programming language but not the other, you need to *prove* that the function you showed is (1) computable in the first programming language and (2) *not computable* in the second programming language.

Exercise 4.11 — Compare IF and NAND. Let IF-CIRC be the programming language where we have the following operations `foo = 0`, `foo = 1`, `foo = IF(cond,yes,no)` (that is, we can use the constants 0 and 1, and the $IF : \{0,1\}^3 \rightarrow \{0,1\}$ function such that $IF(a,b,c)$ equals b if $a = 1$ and equals c if $a = 0$). Compare the power of the NAND-CIRC programming language and the IF-CIRC programming language.

■

Exercise 4.12 — Compare XOR and NAND. Let XOR-CIRC be the programming language where we have the following operations `foo = XOR(bar,blah)`, `foo = 1` and `bar = 0` (that is, we can use the constants 0, 1 and the *XOR* function that maps $a, b \in \{0,1\}^2$ to $a + b \mod 2$). Compare the power of the NAND-CIRC programming language and the XOR-CIRC programming language. See footnote for hint.[7]

Exercise 4.13 — Circuits for majority. Prove that there is some constant c such that for every $n > 1$, $MAJ_n \in SIZE_n(cn)$ where $MAJ_n : \{0,1\}^n \to \{0,1\}$ is the majority function on n input bits. That is $MAJ_n(x) = 1$ iff $\sum_{i=0}^{n-1} x_i > n/2$. See footnote for hint.[8]

Exercise 4.14 — Circuits for threshold. Prove that there is some constant c such that for every $n > 1$, and integers $a_0, \ldots, a_{n-1}, b \in \{-2^n, -2^n + 1, \ldots, -1, 0, +1, \ldots, 2^n\}$, there is a NAND circuit with at most n^c gates that computes the *threshold* function $f_{a_0,\ldots,a_{n-1},b} : \{0,1\}^n \to \{0,1\}$ that on input $x \in \{0,1\}^n$ outputs 1 if and only if $\sum_{i=0}^{n-1} a_i x_i > b$.

4.8 BIBLIOGRAPHICAL NOTES

See Jukna's and Wegener's books [Juk12; Weg87] for much more extensive discussion on circuits. Shannon showed that every Boolean function can be computed by a circuit of exponential size [Sha38]. The improved bound of $c \cdot 2^n/n$ (with the optimal value of c for many bases) is due to Lupanov [Lup58]. An exposition of this for the case of NAND (where $c = 1$) is given in Chapter 4 of his book [Lup84]. (Thanks to Sasha Golovnev for tracking down this reference!)

The concept of "syntactic sugar" is also known as "macros" or "meta-programming" and is sometimes implemented via a preprocessor or macro language in a programming language or a text editor. One modern example is the Babel JavaScript syntax transformer, that converts JavaScript programs written using the latest features into a format that older Browsers can accept. It even has a plug-in architecture, that allows users to add their own syntactic sugar to the language.

[8] One approach to solve this is using recursion and the so-called Master Theorem.

Learning Objectives:

- See one of the most important concepts in computing: duality between code and data.
- Build up comfort in moving between different representations of programs.
- Follow the construction of a "universal circuit evaluator" that can evaluate other circuits given their representation.
- See major result that complements the result of the last chapter: some functions require an *exponential* number of gates to compute.
- Discussion of *Physical extended Church-Turing thesis* stating that Boolean circuits capture *all* feasible computation in the physical world, and its physical and philosophical implications.

5
Code as data, data as code

> *"The term code script is, of course, too narrow. The chromosomal structures are at the same time instrumental in bringing about the development they foreshadow. They are law-code and executive power - or, to use another simile, they are architect's plan and builder's craft - in one."* , Erwin Schrödinger, 1944.

> *"A mathematician would hardly call a correspondence between the set of 64 triples of four units and a set of twenty other units,"universal", while such correspondence is, probably, the most fundamental general feature of life on Earth"*, Misha Gromov, 2013

A program is simply a sequence of symbols, each of which can be encoded as a string of 0's and 1's using (for example) the ASCII standard. Therefore we can represent every NAND-CIRC program (and hence also every Boolean circuit) as a binary string. This statement seems obvious but it is actually quite profound. It means that we can treat circuits or NAND-CIRC programs both as instructions to carrying computation and also as *data* that could potentially be used as *inputs* to other computations.

Big Idea 6 A *program* is a piece of text, and so it can be fed as input to other programs.

This correspondence between *code* and *data* is one of the most fundamental aspects of computing. It underlies the notion of *general purpose* computers, that are not pre-wired to compute only one task, and also forms the basis of our hope for obtaining *general* artificial intelligence. This concept finds immense use in all areas of computing, from scripting languages to machine learning, but it is fair to say that we haven't yet fully mastered it. Many security exploits involve cases such as "buffer overflows" when attackers manage to inject code where the system expected only "passive" data (see Fig. 5.1). The relation between code and data reaches beyond the realm of electronic

computers. For example, DNA can be thought of as both a program and data (in the words of Schrödinger, who wrote before the discovery of DNA's structure a book that inspired Watson and Crick, DNA is both "architect's plan and builder's craft").

This chapter: A non-mathy overview

In this chapter, we will begin to explore some of the many applications of the correspondence between code and data. We start by using the representation of programs/circuits as strings to *count* the number of programs/circuits up to a certain size, and use that to obtain a counterpart to the result we proved in Chapter 4. There we proved that *every* function can be computed by a circuit, but that circuit could be exponentially large (see Theorem 4.16 for the precise bound) In this chapter we will prove that there are *some* functions for which we cannot do better: the *smallest* circuit that computes them is exponentially large.

We will also use the notion of representing programs/circuits as strings to show the existence of a "universal circuit" - a circuit that can evaluate other circuits. In programming languages, this is known as a "meta circular evaluator" - a program in a certain programming language that can evaluate other programs in the same language. These results do have an important restriction: the universal circuit will have to be of bigger size than the circuits it evaluates. We will show how to get around this restriction in Chapter 7 where we introduce *loops* and *Turing machines*.

See Fig. 5.2 for an overview of the results of this chapter.

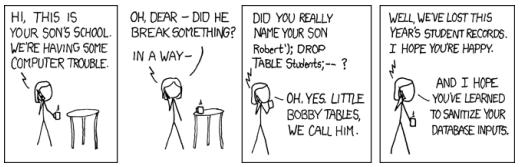


Figure 5.1: As illustrated in this xkcd cartoon, many exploits, including buffer overflow, SQL injections, and more, utilize the blurry line between "active programs" and "static strings".

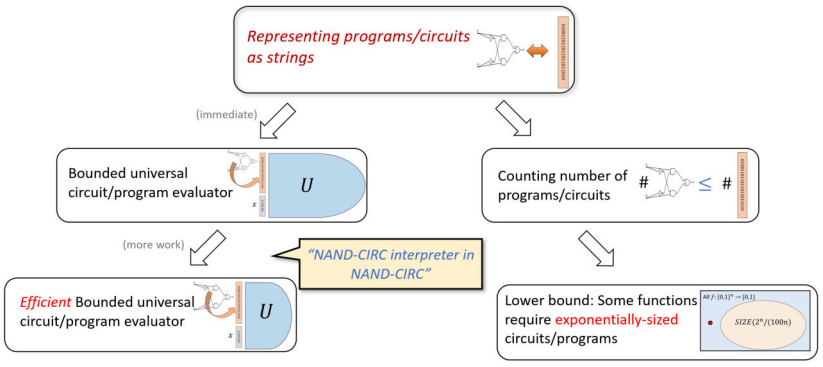


Figure 5.2: Overview of the results in this chapter. We use the representation of programs/circuits as strings to derive two main results. First we show the existence of a universal program/circuit, and in fact (with more work) the existence of such a program/circuit whose size is at most polynomial in the size of the program/circuit it evaluates. We then use the string representation to *count* the number of programs/circuits of a given size, and use that to establish that *some* functions require an *exponential* number of lines/gates to compute.

5.1 REPRESENTING PROGRAMS AS STRINGS

We can represent programs or circuits as strings in a myriad of ways. For example, since Boolean circuits are labeled directed acyclic graphs, we can use the *adjacency matrix* or *adjacency list* representations for them. However, since the code of a program is ultimately just a sequence of letters and symbols, arguably the conceptually simplest representation of a program is as such a sequence. For example, the following NAND-CIRC program P

```
temp_0 = NAND(X[0],X[1])
temp_1 = NAND(X[0],temp_0)
temp_2 = NAND(X[1],temp_0)
Y[0] = NAND(temp_1,temp_2)
```

is simply a string of 107 symbols which include lower and upper case letters, digits, the underscore character _ and equality sign =, punctuation marks such as "(","),"," ,", spaces, and "new line" markers (often denoted as "\n" or "↵"). Each such symbol can be encoded as a string of 7 bits using the ASCII encoding, and hence the program P can be encoded as a string of length $7 \cdot 107 = 749$ bits.

Nothing in the above discussion was specific to the program P, and hence we can use the same reasoning to prove that *every* NAND-CIRC program can be represented as a string in $\{0,1\}^*$. In fact, we can do a bit better. Since the names of the working variables of a NAND-CIRC program do not affect its functionality, we can always transform a program to have the form of P' where all variables apart from the inputs and outputs have the form `temp_0`, `temp_1`, `temp_2`, etc.. Moreover, if the program has s lines, then we will never need to use an index larger than $3s$ (since each line involves at most three variables), and similarly the indices of the input and output variables will all be at most $3s$. Since a number between 0 and $3s$ can be expressed using at most $\lceil \log_{10}(3s+1) \rceil = O(\log s)$ digits, each line in the program (which has the form `foo = NAND(bar,blah)`), can be represented using $O(1) + O(\log s) = O(\log s)$ symbols, each of which can be represented by 7 bits. Hence an s line program can be represented as a string of $O(s \log s)$ bits, resulting in the following theorem:

Theorem 5.1 — Representing programs as strings. There is a constant c such that for $f \in SIZE(s)$, there exists a program P computing f whose string representation has length at most $cs \log s$.

P

We omit the formal proof of Theorem 5.1 but please make sure that you understand why it follows from the reasoning above.

5.2 COUNTING PROGRAMS, AND LOWER BOUNDS ON THE SIZE OF NAND-CIRC PROGRAMS

One consequence of the representation of programs as strings is that the number of programs of certain length is bounded by the number of strings that represent them. This has consequences for the sets $SIZE_{n,m}(s)$ that we defined in Section 4.6.

Theorem 5.2 — Counting programs. For every $s, n, m \in \mathbb{N}$,

$$|SIZE_{n,m}(s)| \leq 2^{O(s \log s)}.$$

That is, there are at most $2^{O(s \log s)}$ functions computed by NAND-CIRC programs of at most s lines. [1]

[1] The implicit constant in the $O(\cdot)$ notation is smaller than 10. That is, for all sufficiently large s, $|SIZE_{n,m}(s)| < 2^{10 s \log s}$, see Remark 5.4. As discussed in Section 1.7, we use the bound 10 simply because it is a round number.

Proof. For any $n, m \in \mathbb{N}$, we will show a one-to-one map E from $SIZE_{n,m}(s)$ to the set of strings of length $cs \log s$ for some constant c. This will conclude the proof, since it implies that $|SIZE_{n,m}(s)|$ is smaller than the size of the set of all strings of length at most $\ell = cs \log s$. The size of the latter set is $1 + 2 + 4 + \cdots + 2^{\ell} = 2^{\ell+1} - 1$ by the formula for sums of geometric progressions.

The map E will simply map f to the representation of the smallest program computing f. Since $f \in SIZE_{n,m}(s)$, there is a program P of at most s lines that can be represented using a string of length at most $cs \log s$ by Theorem 5.1. Moreover, the map $f \mapsto E(f)$ is one to one, since for every distinct $f, f' : \{0,1\}^n \to \{0,1\}^m$ there must exist some input $x \in \{0,1\}^n$ on which $f(x) \neq f'(x)$. This means that the programs that compute f and f' respectively cannot be identical. ■

Theorem 5.2 has an important corollary. The number of functions that can be computed using small circuits/programs is much smaller than the total number of functions, and hence there exist functions that require very large (in fact *exponentially large*) circuits to compute. To see why this is the case, note that a function mapping $\{0,1\}^2$ to $\{0,1\}$ can be identified with the list of its four values on the inputs $00, 01, 10, 11$. A function mapping $\{0,1\}^3$ to $\{0,1\}$ can be identified with the list of its eight values on the inputs $000, 001, 010, 011, 100, 101, 110, 111$. More generally, every function $F : \{0,1\}^n \to \{0,1\}$ can be identified with the list of its 2^n values on

the inputs $\{0,1\}^n$. Hence the number of functions mapping $\{0,1\}^n$ to $\{0,1\}$ is equal to the number of possible 2^n length lists of values which is exactly 2^{2^n}. Note that this is *double exponential* in n, and hence even for small values of n (e.g., $n = 10$) the number of functions from $\{0,1\}^n$ to $\{0,1\}$ is truly astronomical.[2] As mentioned, this yields the following corollary:

Theorem 5.3 — Counting argument lower bound. There is a constant $\delta > 0$, such that for every sufficiently large n, there is a function $f : \{0,1\}^n \to \{0,1\}$ such that $f \notin SIZE_n\left(\frac{\delta 2^n}{n}\right)$. That is, the shortest NAND-CIRC program to compute f requires more than $\delta \cdot 2^n/n$ lines. [3]

Proof. The proof is simple. If we let c be the constant such that $|SIZE_n(s)| \leq 2^{cs \log s}$ and $\delta = 1/c$, then setting $s = \delta 2^n/n$ we see that

$$|SIZE_n(\tfrac{\delta 2^n}{n})| \leq 2^{c\frac{\delta 2^n}{n} \log s} < 2^{c\delta 2^n} = 2^{2^n}$$

using the fact that since $s < 2^n$, $\log s < n$ and $\delta = 1/c$. But since $|SIZE_n(s)|$ is smaller than the total number of functions mapping n bits to 1 bit, there must be at least one such function not in $SIZE_n(s)$, which is what we needed to prove. ■

We have seen before that *every* function mapping $\{0,1\}^n$ to $\{0,1\}$ can be computed by an $O(2^n/n)$ line program. Theorem 5.3 shows that this is tight in the sense that some functions do require such an astronomical number of lines to compute.

Big Idea 7 Some functions $f : \{0,1\}^n \to \{0,1\}$ *cannot* be computed by a Boolean circuit using fewer than exponential (in n) number of gates.

In fact, as we explore in the exercises, this is the case for *most* functions. Hence functions that can be computed in a small number of lines (such as addition, multiplication, finding short paths in graphs, or even the *EVAL* function) are the exception, rather than the rule.

R

Remark 5.4 — More efficient representation (advanced, optional). The ASCII representation is not the shortest representation for NAND-CIRC programs. NAND-CIRC programs are equivalent to circuits with NAND gates, which means that a NAND-CIRC program of s lines, n inputs, and m outputs can be represented by a labeled directed graph of $s + n$ vertices, of which n

[2] "Astronomical" here is an understatement: there are much fewer than $2^{2^{10}}$ stars, or even particles, in the observable universe.

[3] The constant δ is at least 0.1 and in fact, can be improved to be arbitrarily close to $1/2$, see Exercise 5.7.

have in-degree zero, and the s others have in-degree at most two. Using the adjacency matrix representation for such graphs, we can reduce the implicit constant in Theorem 5.2 to be arbitrarily close to 5, see Exercise 5.6.

5.2.1 Size hierarchy theorem (optional)

By Theorem 4.15 the class $SIZE_n(10 \cdot 2^n/n)$ contains *all* functions from $\{0,1\}^n$ to $\{0,1\}$, while by Theorem 5.3, there is *some* function $f : \{0,1\}^n \to \{0,1\}$ that is *not contained* in $SIZE_n(0.1 \cdot 2^n/n)$. In other words, for every sufficiently large n,

$$SIZE_n\left(0.1\tfrac{2^n}{n}\right) \subsetneq SIZE_n\left(10\tfrac{2^n}{n}\right) .$$

It turns out that we can use Theorem 5.3 to show a more general result: whenever we increase our "budget" of gates we can compute new functions.

Theorem 5.5 — Size Hierarchy Theorem. For every sufficiently large n and $10n < s < 0.1 \cdot 2^n/n$,

$$SIZE_n(s) \subsetneq SIZE_n(s + 10n) .$$

Proof Idea:

To prove the theorem we need to find a function $f : \{0,1\}^n \to \{0,1\}$ such that f *can* be computed by a circuit of $s + 10n$ gates but it *cannot* be computed by a circuit of s gates. We will do so by coming up with a sequence of functions $f_0, f_1, f_2, \ldots, f_N$ with the following properties: **(1)** f_0 *can* be computed by a circuit of at most $10n$ gates, **(2)** f_N *cannot* be computed by a circuit of $0.1 \cdot 2^n/n$ gates, and **(3)** for every $i \in \{0, \ldots, N\}$, if f_i can be computed by a circuit of size s, then f_{i+1} can be computed by a circuit of size at most $s+10n$. Together these properties imply that if we let i be the smallest number such that $f_i \notin SIZE_n(s)$, then since $f_{i-1} \in SIZE_n(s)$ it must hold that $f_i \in SIZE_n(s+10n)$ which is what we need to prove. See Fig. 5.4 for an illustration.

⋆

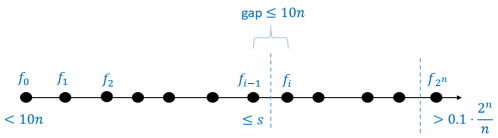


Figure 5.4: We prove Theorem 5.5 by coming up with a list $f_0, \ldots, f_{2^n}$ of functions such that f_0 is the all zero function, f_{2^n} is a function (obtained from Theorem 5.3) outside of $SIZE_n(0.1 \cdot 2^n/n)$ and such that f_{i-1} and f_i differ by one another on at most one input. We can show that for every i, the number of gates to compute f_i is at most $10n$ larger than the number of gates to compute f_{i-1} and so if we let i be the smallest number such that $f_i \notin SIZE_n(s)$, then $f_i \in SIZE_n(s + 10n)$.

Proof of Theorem 5.5. Let $f^* : \{0,1\}^n \to \{0,1\}$ be the function (whose existence we are guaranteed by Theorem 5.3) such that $f^* \notin SIZE_n(0.1 \cdot 2^n/n)$. We define the functions $f_0, f_1, \ldots, f_{2^n}$ mapping $\{0,1\}^n$ to $\{0,1\}$ as follows. For every $x \in \{0,1\}^n$, if $lex(x) \in \{0, 1, \ldots, 2^n - 1\}$ is x's order in the lexicographical order then

$$f_i(x) = \begin{cases} f^*(x) & lex(x) < i \\ 0 & \text{otherwise} \end{cases} .$$

The function f_0 is simply the constant zero function, while the function f_{2^n} is equal to f^*. Moreover, for every $i \in [2^n]$, the functions f_i and f_{i+1} differ on at most one input (i.e., the input $x \in \{0,1\}^n$ such that $lex(x) = i$). Let $10n < s < 0.1 \cdot 2^n/n$, and let i be the first index such that $f_i \notin SIZE_n(s)$. Since $f_{2^n} = f^* \notin SIZE_n(0.1 \cdot 2^n/n)$ there must exist such an index i, and moreover $i > 0$ since the constant zero function is a member of $SIZE_n(10n)$.

By our choice of i, f_{i-1} is a member of $SIZE_n(s)$. To complete the proof, we need to show that $f_i \in SIZE_n(s + 10n)$. Let x^* be the string such that $lex(x^*) = i$ and let $b \in \{0,1\}$ be the value of $f^*(x^*)$. Then we can define f_i also as follows

$$f_i(x) = \begin{cases} b & x = x^* \\ f_i(x) & x \neq x^* \end{cases}$$

or in other words

$$f_i(x) = f_{i-1}(x) \wedge EQUAL(x^*, x) \ \vee \ b \wedge \neg EQUAL(x^*, x)$$

where $EQUAL : \{0,1\}^{2n} \to \{0,1\}$ is the function that maps $x, x' \in \{0,1\}^n$ to 1 if they are equal and to 0 otherwise. Since (by our choice of i), f_{i-1} can be computed using at most s gates and (as can be easily verified) that $EQUAL \in SIZE_n(9n)$, we can compute f_i using at most $s + 9n + O(1) \leq s + 10n$ gates which is what we wanted to prove.

■

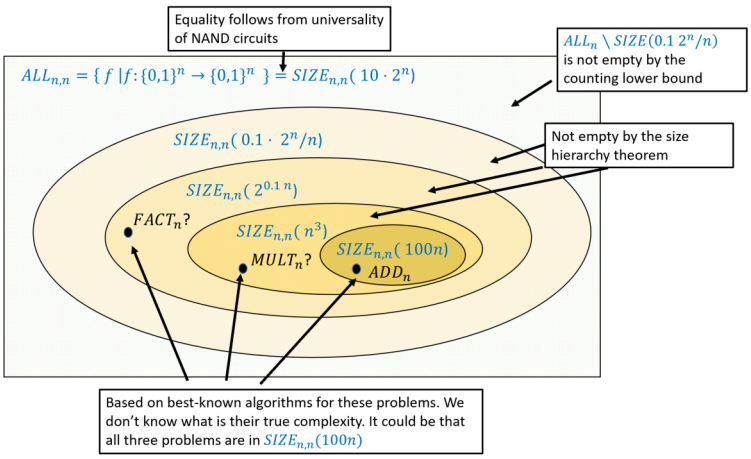


Figure 5.5: An illustration of some of what we know about the size complexity classes (not to scale!). This figure depicts classes of the form $SIZE_{n,n}(s)$ but the state of affairs for other size complexity classes such as $SIZE_{n,1}(s)$ is similar. We know by Theorem 4.12 (with the improvement of Section 4.4.2) that all functions mapping n bits to n bits can be computed by a circuit of size $c \cdot 2^n$ for $c \leq 10$, while on the other hand the counting lower bound (Theorem 5.3, see also Exercise 5.4) shows that *some* such functions will require $0.1 \cdot 2^n$, and the size hierarchy theorem (Theorem 5.5) shows the existence of functions in $SIZE_n(S) \setminus SIZE_n(s)$ whenever $s = o(S)$, see also Exercise 5.5. We also consider some specific examples: addition of two $n/2$ bit numbers can be done in $O(n)$ lines, while we don't know of such a program for *multiplying* two n bit numbers, though we do know it can be done in $O(n^2)$ and in fact even better size. In the above, $FACTOR_n$ corresponds to the inverse problem of multiplying- finding the *prime factorization* of a given number. At the moment we do not know of any circuit a polynomial (or even sub-exponential) number of lines that can compute $FACTOR_n$.

R

Remark 5.6 — Explicit functions. While the size hierarchy theorem guarantees that there exists *some* function that *can* be computed using, for example, n^2 gates, but

not using $100n$ gates, we do not know of any explicit example of such a function. While we suspect that integer multiplication is such an example, we do not have any proof that this is the case.

5.3 THE TUPLES REPRESENTATION

ASCII is a fine presentation of programs, but for some applications it is useful to have a more concrete representation of NAND-CIRC programs. In this section we describe a particular choice, that will be convenient for us later on. A NAND-CIRC program is simply a sequence of lines of the form

```
blah = NAND(baz,boo)
```

There is of course nothing special about the particular names we use for variables. Although they would be harder to read, we could write all our programs using only working variables such as `temp_0`, `temp_1` etc. Therefore, our representation for NAND-CIRC programs ignores the actual names of the variables, and just associate a *number* with each variable. We encode a *line* of the program as a triple of numbers. If the line has the form `foo = NAND(bar,blah)` then we encode it with the triple (i, j, k) where i is the number corresponding to the variable `foo` and j and k are the numbers corresponding to `bar` and `blah` respectively.

More concretely, we will associate every variable with a number in the set $[t] = \{0, 1, \ldots, t-1\}$. The first n numbers $\{0, \ldots, n-1\}$ correspond to the *input* variables, the last m numbers $\{t-m, \ldots, t-1\}$ correspond to the *output* variables, and the intermediate numbers $\{n, \ldots, t-m-1\}$ correspond to the remaining "workspace" variables. Formally, we define our representation as follows:

Definition 5.7 — List of tuples representation. Let P be a NAND-CIRC program of n inputs, m outputs, and s lines, and let t be the number of distinct variables used by P. The *list of tuples representation of* P is the triple (n, m, L) where L is a list of triples of the form (i, j, k) for $i, j, k \in [t]$.

We assign a number for each variable of P as follows:

- For every $i \in [n]$, the variable `X[`i`]` is assigned the number i.
- For every $j \in [m]$, the variable `Y[`j`]` is assigned the number $t - m + j$.

- Every other variable is assigned a number in $\{n, n+1, \ldots, t-m-1\}$ in the order in which the variable appears in the program P.

The list of tuples representation is our default choice for representing NAND-CIRC programs. Since "list of tuples representation" is a bit of a mouthful, we will often call it simply "the representation" for a program P. Sometimes, when the number n of inputs and number m of outputs are known from the context, we will simply represent a program as the list L instead of the triple (n, m, L).

■ **Example 5.8 — Representing the XOR program.** Our favorite NAND-CIRC program, the program

```
u = NAND(X[0],X[1])
v = NAND(X[0],u)
w = NAND(X[1],u)
Y[0] = NAND(v,w)
```

computing the XOR function is represented as the tuple $(2, 1, L)$ where $L = ((2, 0, 1), (3, 0, 2), (4, 1, 2), (5, 3, 4))$. That is, the variables `X[0]` and `X[1]` are given the indices 0 and 1 respectively, the variables `u`,`v`,`w` are given the indices $2, 3, 4$ respectively, and the variable `Y[0]` is given the index 5.

Transforming a NAND-CIRC program from its representation as code to the representation as a list of tuples is a fairly straightforward programming exercise, and in particular can be done in a few lines of *Python*.[4] The list-of-tuples representation loses information such as the particular names we used for the variables, but this is OK since these names do not make a difference to the functionality of the program.

[4] If you're curious what these few lines are, see our GitHub repository.

5.3.1 From tuples to strings

If P is a program of size s, then the number t of variables is at most $3s$ (as every line touches at most three variables). Hence we can encode every variable index in $[t]$ as a string of length $\ell = \lceil \log(3s) \rceil$, by adding leading zeroes as needed. Since this is a fixed-length encoding, it is prefix free, and so we can encode the list L of s triples (corresponding to the encoding of the s lines of the program) as simply the string of length $3\ell s$ obtained by concatenating all of these encodings.

We define $S(s)$ to be the length of the string representing the list L corresponding to a size s program. By the above we see that

$$S(s) = 3s\lceil \log(3s) \rceil \ . \tag{5.1}$$

We can represent $P = (n, m, L)$ as a string by prepending a prefix free representation of n and m to the list L. Since $n, m \leq 3s$ (a pro-

gram must touch at least once all its input and output variables), those prefix free representations can be encoded using strings of length $O(\log s)$. In particular, every program P of at most s lines can be represented by a string of length $O(s \log s)$. Similarly, every circuit C of at most s gates can be represented by a string of length $O(s \log s)$ (for example by translating C to the equivalent program P).

5.4 A NAND-CIRC INTERPRETER IN NAND-CIRC

Since we can represent programs as strings, we can also think of a program as an input to a function. In particular, for every natural number $s, n, m > 0$ we define the function $EVAL_{s,n,m} : \{0,1\}^{S(s)+n} \to \{0,1\}^m$ as follows:

$$EVAL_{s,n,m}(px) = \begin{cases} P(x) & p \in \{0,1\}^{S(s)} \text{ represents a size-}s \text{ program } P \text{ with } n \text{ inputs and } m \text{ outputs} \\ 0^m & \text{otherwise} \end{cases} \tag{5.2}$$

where $S(s)$ is defined as in (5.1) and we use the concrete representation scheme described in Section 5.1.

That is, $EVAL_{s,n,m}$ takes as input the concatenation of two strings: a string $p \in \{0,1\}^{S(s)}$ and a string $x \in \{0,1\}^n$. If p is a string that represents a list of triples L such that (n, m, L) is a list-of-tuples representation of a size-s NAND-CIRC program P, then $EVAL_{s,n,m}(px)$ is equal to the evaluation $P(x)$ of the program P on the input x. Otherwise, $EVAL_{s,n,m}(px)$ equals 0^m (this case is not very important: you can simply think of 0^m as some "junk value" that indicates an error).

Take-away points. The fine details of $EVAL_{s,n,m}$'s definition are not very crucial. Rather, what you need to remember about $EVAL_{s,n,m}$ is that:

- $EVAL_{s,n,m}$ is a finite function taking a string of fixed length as input and outputting a string of fixed length as output.
- $EVAL_{s,n,m}$ is a single function, such that computing $EVAL_{s,n,m}$ allows to evaluate *arbitrary* NAND-CIRC programs of a certain length on *arbitrary* inputs of the appropriate length.
- $EVAL_{s,n,m}$ is a *function*, not a *program* (recall the discussion in Section 3.7.2). That is, $EVAL_{s,n,m}$ is a *specification* of what output is associated with what input. The existence of a *program* that computes $EVAL_{s,n,m}$ (i.e., an *implementation* for $EVAL_{s,n,m}$) is a separate fact, which needs to be established (and which we will do in Theorem 5.9, with a more efficient program shown in Theorem 5.10).

One of the first examples of *self circularity* we will see in this book is the following theorem, which we can think of as showing a "NAND-CIRC interpreter in NAND-CIRC":

Theorem 5.9 — Bounded Universality of NAND-CIRC programs. For every $s, n, m \in \mathbb{N}$ with $s \geq m$ there is a NAND-CIRC program $U_{s,n,m}$ that computes the function $EVAL_{s,n,m}$.

That is, the NAND-CIRC program $U_{s,n,m}$ takes the description of *any other NAND-CIRC program* P (of the right length and inputs/outputs) and *any input* x, and computes the result of evaluating the program P on the input x. Given the equivalence between NAND-CIRC programs and Boolean circuits, we can also think of $U_{s,n,m}$ as a circuit that takes as input the description of other circuits and their inputs, and returns their evaluation, see Fig. 5.6. We call this NAND-CIRC program $U_{s,n,m}$ that computes $EVAL_{s,n,m}$ a *bounded universal program* (or a *universal circuit*, see Fig. 5.6). "Universal" stands for the fact that this is a *single program* that can evaluate *arbitrary* code, where "bounded" stands for the fact that $U_{s,n,m}$ only evaluates programs of bounded size. Of course this limitation is inherent for the NAND-CIRC programming language, since a program of s lines (or, equivalently, a circuit of s gates) can take at most $2s$ inputs. Later, in Chapter 7, we will introduce the concept of *loops* (and the model of *Turing machines*), that allow to escape this limitation.

Proof. Theorem 5.9 is an important result, but it is actually not hard to prove. Specifically, since $EVAL_{s,n,m}$ is a finite function Theorem 5.9 is an immediate corollary of Theorem 4.12, which states that *every* finite function can be computed by *some* NAND-CIRC program. ■

P

Theorem 5.9 is simple but important. Make sure you understand what this theorem means, and why it is a corollary of Theorem 4.12.

5.4.1 Efficient universal programs

Theorem 5.9 establishes the existence of a NAND-CIRC program for computing $EVAL_{s,n,m}$, but it provides no explicit bound on the size of this program. Theorem 4.12, which we used to prove Theorem 5.9, guarantees the existence of a NAND-CIRC program whose size can be as large as *exponential* in the length of its input. This would mean that even for moderately small values of s, n, m (for example $n = 100, s = 300, m = 1$), computing $EVAL_{s,n,m}$ might require a NAND program with more lines than there are atoms in the observable universe! Fortunately, we can do much better than that. In fact, for every s, n, m there exists a NAND-CIRC program for comput-

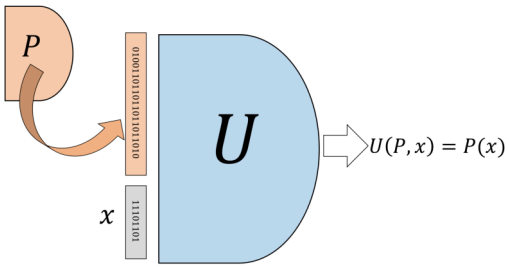


Figure 5.6: A *universal circuit* U is a circuit that gets as input the description of an arbitrary (smaller) circuit P as a binary string, and an input x, and outputs the string $P(x)$ which is the evaluation of P on x. We can also think of U as a straight-line program that gets as input the code of a straight-line program P and an input x, and outputs $P(x)$.

ing $EVAL_{s,n,m}$ with size that is *polynomial* in its input length. This is shown in the following theorem.

Theorem 5.10 — Efficient bounded universality of NAND-CIRC programs. For every $s, n, m \in \mathbb{N}$ there is a NAND-CIRC program of at most $O(s^2 \log s)$ lines that computes the function $EVAL_{s,n,m} : \{0,1\}^{S+n} \to \{0,1\}^m$ defined above (where S is the number of bits needed to represent programs of s lines).

If you haven't done so already, now might be a good time to review O notation in Section 1.4.8. In particular, an equivalent way to state Theorem 5.10 is that it says that there *exists* some number $c > 0$ such that *for every* $s, n, m \in \mathbb{N}$, there exists a NAND-CIRC program P of at most $cs^2 \log s$ lines that computes the function $EVAL_{s,n,m}$.

Unlike Theorem 5.9, Theorem 5.10 is not a trivial corollary of the fact that every finite function can be computed by some circuit. Proving Theorem 5.10 requires us to present a concrete NAND-CIRC program for computing the function $EVAL_{s,n,m}$. We will do so in several stages.

1. First, we will describe the algorithm to evaluate $EVAL_{s,n,m}$ in "pseudo code".
2. Then, we will show how we can write a program to compute $EVAL_{s,n,m}$ in *Python*. We will not use much about Python, and a reader that has familiarity with programming in any language should be able to follow along.
3. Finally, we will show how we can transform this Python program into a NAND-CIRC program.

This approach yields much more than just proving Theorem 5.10: we will see that it is in fact always possible to transform (loop free) code in high level languages such as Python to NAND-CIRC programs (and hence to Boolean circuits as well).

5.4.2 A NAND-CIRC interpeter in "pseudocode"

To prove Theorem 5.10 it suffices to give a NAND-CIRC program of $O(s^2 \log s)$ lines that can evaluate NAND-CIRC programs of s lines. Let us start by thinking how we would evaluate such programs if we weren't restricted to only performing NAND operations. That is, let us describe informally an *algorithm* that on input n, m, s, a list of triples

L, and a string $x \in \{0,1\}^n$, evaluates the program represented by (n, m, L) on the string x.

It would be highly worthwhile for you to stop here and try to solve this problem yourself. For example, you can try thinking how you would write a program `NANDEVAL(n,m,s,L,x)` that computes this function in the programming language of your choice.

We will now describe such an algorithm. We assume that we have access to a *bit array* data structure that can store for every $i \in [t]$ a bit $T_i \in \{0,1\}$. Specifically, if `Table` is a variable holding this data structure, then we assume we can perform the operations:

- `GET(Table,i)` which retrieves the bit corresponding to `i` in `Table`. The value of `i` is assumed to be an integer in $[t]$.
- `Table = UPDATE(Table,i,b)` which updates `Table` so the bit corresponding to `i` is now set to `b`. The value of `i` is assumed to be an integer in $[t]$ and `b` is a bit in $\{0,1\}$.

Algorithm 5.11 — Eval NAND-CIRC programs.

Input: Numbers n, m, s and $t \leq 3s$, as well as a list L of s triples of numbers in $[t]$, and a string $x \in \{0,1\}^n$.
Output: Evaluation of the program represented by (n, m, L) on the
Input: $x \in \{0,1\}^n$.

```
Let Vartable be table of size t
for i in [n] do
    Vartable = UPDATE(Vartable,i,x_i)
end for
for (i,j,k) in L do
    a ← GET(Vartable,j)
    b ← GET(Vartable,k)
    Vartable = UPDATE(Vartable,i,NAND(a,b))
end for
for j in [m] do
    y_j ← GET(Vartable,t − m + j)
end for
return y_0, ..., y_{m−1}
```

Algorithm 5.11 evaluates the program given to it as input one line at a time, updating the `Vartable` table to contain the value of each

variable. At the end of the execution it outputs the variables at positions $t-m, t-m+1, \ldots, t-1$ which correspond to the input variables.

5.4.3 A NAND interpreter in Python

To make things more concrete, let us see how we implement Algorithm 5.11 in the *Python* programming language. (There is nothing special about Python. We could have easily presented a corresponding function in JavaScript, C, OCaml, or any other programming language.) We will construct a function `NANDEVAL` that on input n, m, L, x will output the result of evaluating the program represented by (n, m, L) on x. To keep things simple, we will not worry about the case that L does not represent a valid program of n inputs and m outputs. The code is presented in Fig. 5.7.

Accessing an element of the array `Vartable` at a given index takes a constant number of basic operations. Hence (since $n, m \leq s$ and $t \leq 3s$), the program above will use $O(s)$ basic operations.[5]

[5] Python does not distinguish between lists and arrays, but allows constant time random access to an indexed elements to both of them. One could argue that if we allowed programs of truly unbounded length (e.g., larger than 2^{64}) then the price would not be constant but logarithmic in the length of the array/lists, but the difference between $O(s)$ and $O(s \log s)$ will not be important for our discussions.

5.4.4 Constructing the NAND-CIRC interpreter in NAND-CIRC

We now turn to describing the proof of Theorem 5.10. To prove the theorem it is not enough to give a Python program. Rather, we need to show how we compute the function $EVAL_{s,n,m}$ using a *NAND-CIRC program*. In other words, our job is to transform, for every s, n, m, the Python code of Section 5.4.3 to a NAND-CIRC program $U_{s,n,m}$ that computes the function $EVAL_{s,n,m}$.

P

Before reading further, try to think how *you* could give a "constructive proof" of Theorem 5.10. That is, think of how you would write, in the programming language of your choice, a function `universal(s,n,m)` that on input s, n, m outputs the code for the NAND-CIRC program $U_{s,n,m}$ such that $U_{s,n,m}$ computes $EVAL_{s,n,m}$. There is a subtle but crucial difference between this function and the Python `NANDEVAL` program described above. Rather than actually evaluating a given program P on some input w, the function `universal` should output the *code* of a NAND-CIRC program that computes the map $(P, x) \mapsto P(x)$.

Our construction will follow very closely the Python implementation of `EVAL` above. We will use variables `Vartable[0]`,...,`Vartable[`$2^{\ell}-1$`]`, where $\ell = \lceil \log 3s \rceil$ to store our variables. However, NAND doesn't have integer-valued variables, so we cannot write code such as `Vartable[i]` for some variable `i`. However, we *can* implement the function `GET(Vartable,i)` that outputs the `i`-th bit of the array `Vartable`. Indeed, this is nothing but the function $LOOKUP_{\ell}$ that we have seen in Theorem 4.10!

Figure 5.7: Code for evaluating a NAND-CIRC program given in the list-of-tuples representation

```
def NANDEVAL(n,m,L,X):
    # Evaluate a NAND-CIRC program from list of tuple representation.
    s = len(L) # num of lines
    t = max(max(a,b,c) for (a,b,c) in L)+1 # max index in L + 1
    Vartable = [0] * t # initialize array

    # helper functions
    def GET(V,i): return V[i]
    def UPDATE(V,i,b):
        V[i]=b
        return V

    # load input values to Vartable:
    for i in range(n):
        Vartable = UPDATE(Vartable,i,X[i])

    # Run the program
    for (i,j,k) in L:
        a = GET(Vartable,j)
        b = GET(Vartable,k)
        c = NAND(a,b)
        Vartable = UPDATE(Vartable,i,c)

    # Return outputs Vartable[t-m], Vartable[t-m+1],....,Vartable[t-1]
    return [GET(Vartable,t-m+j) for j in range(m)]

# Test on XOR (2 inputs, 1 output)
L = ((2, 0, 1), (3, 0, 2), (4, 1, 2), (5, 3, 4))
print(NANDEVAL(2,1,L,(0,1))) # XOR(0,1)
# [1]
print(NANDEVAL(2,1,L,(1,1))) # XOR(1,1)
# [0]
```

P Please make sure that you understand why GET and $LOOKUP_\ell$ are the same function.

We saw that we can compute $LOOKUP_\ell$ in time $O(2^\ell) = O(s)$ for our choice of ℓ.

For every ℓ, let $UPDATE_\ell : \{0,1\}^{2^\ell+\ell+1} \to \{0,1\}^{2^\ell}$ correspond to the UPDATE function for arrays of length 2^ℓ. That is, on input $V \in \{0,1\}^{2^\ell}$, $i \in \{0,1\}^\ell$, $b \in \{0,1\}$, $UPDATE_\ell(V,b,i)$ is equal to $V' \in \{0,1\}^{2^\ell}$ such that

$$V'_j = \begin{cases} V_j & j \neq i \\ b & j = i \end{cases}$$

where we identify the string $i \in \{0,1\}^\ell$ with a number in $\{0,\ldots,2^\ell-1\}$ using the binary representation. We can compute $UPDATE_\ell$ using an $O(2^\ell \ell) = (s \log s)$ line NAND-CIRC program as follows:

1. For every $j \in [2^\ell]$, there is an $O(\ell)$ line NAND-CIRC program to compute the function $EQUALS_j : \{0,1\}^\ell \to \{0,1\}$ that on input i outputs 1 if and only if i is equal to (the binary representation of) j. (We leave verifying this as Exercise 5.2 and Exercise 5.3.)

2. We have seen that we can compute the function $IF : \{0,1\}^3 \to \{0,1\}$ such that $IF(a,b,c)$ equals b if $a = 1$ and c if $a = 0$.

Together, this means that we can compute UPDATE (using some "syntactic sugar" for bounded length loops) as follows:

```
def UPDATE_ell(V,i,b):
    # Get V[0]...V[2^ell-1], i in {0,1}^ell, b in {0,1}
    # Return NewV[0],...,NewV[2^ell-1]
    # updated array with NewV[i]=b and all
    # else same as V
    for j in range(2**ell): # j = 0,1,2,....,2^ell -1
        a = EQUALS_j(i)
        NewV[j] = IF(a,b,V[j])
    return NewV
```

Since the loop over j in UPDATE is run 2^ℓ times, and computing EQUALS_j takes $O(\ell)$ lines, the total number of lines to compute UPDATE is $O(2^\ell \cdot \ell) = O(s \log s)$. Once we can compute GET and UPDATE, the rest of the implementation amounts to "book keeping" that needs to be done carefully, but is not too insightful, and hence we omit the full details. Since we run GET and UPDATE s times, the total number of lines for computing $EVAL_{s,n,m}$ is $O(s^2) + O(s^2 \log s) = O(s^2 \log s)$. This completes (up to the omitted details) the proof of Theorem 5.10.

R

Remark 5.12 — Improving to quasilinear overhead (advanced optional note). The NAND-CIRC program above is less efficient than its Python counterpart, since NAND does not offer arrays with efficient random access. Hence for example the `LOOKUP` operation on an array of s bits takes $\Omega(s)$ lines in NAND even though it takes $O(1)$ steps (or maybe $O(\log s)$ steps, depending on how we count) in *Python*.

It turns out that it is possible to improve the bound of Theorem 5.10, and evaluate s line NAND-CIRC programs using a NAND-CIRC program of $O(s \log s)$ lines. The key is to consider the description of NAND-CIRC programs as circuits, and in particular as directed acyclic graphs (DAGs) of bounded in-degree. A universal NAND-CIRC program U_s for s line programs will correspond to a *universal graph* H_s for such s vertex DAGs. We can think of such a graph U_s as fixed "wiring" for a communication network, that should be able to accommodate any arbitrary pattern of communication between s vertices (where this pattern corresponds to an s line NAND-CIRC program). It turns out that such efficient routing networks exist that allow embedding any s vertex circuit inside a universal graph of size $O(s \log s)$, see the bibliographical notes Section 5.9 for more on this issue.

5.5 A PYTHON INTERPRETER IN NAND-CIRC (DISCUSSION)

To prove Theorem 5.10 we essentially translated every line of the Python program for `EVAL` into an equivalent NAND-CIRC snippet. However, none of our reasoning was specific to the particular function *EVAL*. It is possible to translate *every* Python program into an equivalent NAND-CIRC program of comparable efficiency. (More concretely, if the Python program takes $T(n)$ operations on inputs of length at most n then there exists NAND-CIRC program of $O(T(n) \log T(n))$ lines that agrees with the Python program on inputs of length n.) Actually doing so requires taking care of many details and is beyond the scope of this book, but let me try to convince you why you should believe it is possible in principle.

For starters, one can use CPython (the reference implementation for Python), to evaluate every Python program using a `C` program. We can combine this with a C compiler to transform a Python program to various flavors of "machine language". So, to transform a Python program into an equivalent NAND-CIRC program, it is enough to show how to transform a *machine language* program into an equivalent NAND-CIRC program. One minimalistic (and hence convenient) family of machine languages is known as the *ARM architecture* which

powers many mobile devices including essentially all Android devices.[6] There are even simpler machine languages, such as the LEG architecture for which a backend for the LLVM compiler was implemented (and hence can be the target of compiling any of the large and growing list of languages that this compiler supports). Other examples include the TinyRAM architecture (motivated by interactive proof systems that we will discuss in Chapter 22) and the teaching-oriented Ridiculously Simple Computer architecture. Going one by one over the instruction sets of such computers and translating them to NAND snippets is no fun, but it is a feasible thing to do. In fact, ultimately this is very similar to the transformation that takes place in converting our high level code to actual silicon gates that are not so different from the operations of a NAND-CIRC program. Indeed, tools such as MyHDL that transform "Python to Silicon" can be used to convert a Python program to a NAND-CIRC program.

[6] ARM stands for "Advanced RISC Machine" where RISC in turn stands for "Reduced instruction set computer".

The NAND-CIRC programming language is just a teaching tool, and by no means do I suggest that writing NAND-CIRC programs, or compilers to NAND-CIRC, is a practical, useful, or enjoyable activity. What I do want is to make sure you understand why it *can* be done, and to have the confidence that if your life (or at least your grade) depended on it, then you would be able to do this. Understanding how programs in high level languages such as Python are eventually transformed into concrete low-level representation such as NAND is fundamental to computer science.

The astute reader might notice that the above paragraphs only outlined why it should be possible to find for every *particular* Python-computable function f, a *particular* comparably efficient NAND-CIRC program P that computes f. But this still seems to fall short of our goal of writing a "Python interpreter in NAND" which would mean that for every parameter n, we come up with a *single* NAND-CIRC program $UNIV_s$ such that given a description of a Python program P, a particular input x, and a bound T on the number of operations (where the lengths of P and x and the value of T are all at most s) returns the result of executing P on x for at most T steps. After all, the transformation above takes every Python program into a *different* NAND-CIRC program, and so does not yield "one NAND-CIRC program to rule them all" that can evaluate every Python program up to some given complexity. However, we can in fact obtain one NAND-CIRC program to evaluate *arbitrary* Python programs. The reason is that there exists a Python interpreter *in Python*: a Python program U that takes a bit string, interprets it as Python code, and then runs that code. Hence, we only need to show a NAND-CIRC program U^* that computes the same function as the particular Python program U, and this will give us a way to evaluate *all* Python programs.

What we are seeing time and again is the notion of *universality* or *self reference* of computation, which is the sense that all reasonably rich models of computation are expressive enough that they can "simulate themselves". The importance of this phenomenon to both the theory and practice of computing, as well as far beyond it, including the foundations of mathematics and basic questions in science, cannot be overstated.

5.6 THE PHYSICAL EXTENDED CHURCH-TURING THESIS (DISCUSSION)

We've seen that NAND gates (and other Boolean operations) can be implemented using very different systems in the physical world. What about the reverse direction? Can NAND-CIRC programs simulate any physical computer?

We can take a leap of faith and stipulate that Boolean circuits (or equivalently NAND-CIRC programs) do actually encapsulate *every* computation that we can think of. Such a statement (in the realm of infinite functions, which we'll encounter in Chapter 7) is typically attributed to Alonzo Church and Alan Turing, and in that context is known as the *Church-Turing Thesis*. As we will discuss in future lectures, the Church-Turing Thesis is not a mathematical theorem or conjecture. Rather, like theories in physics, the Church-Turing Thesis is about mathematically modeling the real world. In the context of finite functions, we can make the following informal hypothesis or prediction:

"Physical Extended Church-Turing Thesis" (PECTT): *If a function $F : \{0,1\}^n \rightarrow \{0,1\}^m$ can be computed in the physical world using s amount of "physical resources" then it can be computed by a Boolean circuit program of roughly s gates.*

A priori it might seem rather extreme to hypothesize that our meager model of NAND-CIRC programs or Boolean circuits captures all possible physical computation. But yet, in more than a century of computing technologies, no one has yet built any scalable computing device that challenges this hypothesis.

We now discuss the "fine print" of the PECTT in more detail, as well as the (so far unsuccessful) challenges that have been raised against it. There is no single universally-agreed-upon formalization of "roughly s physical resources", but we can approximate this notion by considering the size of any physical computing device and the time it takes to compute the output, and ask that any such device can be simulated by a Boolean circuit with a number of gates that is a polynomial (with not too large exponent) in the size of the system and the time it takes it to operate.

In other words, we can phrase the PECTT as stipulating that any function that can be computed by a device that takes a certain volume V of space and requires t time to complete the computation, must be computable by a Boolean circuit with a number of gates $p(V,t)$ that is polynomial in V and t.

The exact form of the function $p(V,t)$ is not universally agreed upon but it is generally accepted that if $f : \{0,1\}^n \rightarrow \{0,1\}$ is an *exponentially hard* function, in the sense that it has no NAND-CIRC program of fewer than, say, $2^{n/2}$ lines, then a demonstration of a physical device that can compute in the real world f for moderate input lengths (e.g., $n = 500$) would be a violation of the PECTT.

R

Remark 5.13 — Advanced note: making PECTT concrete (advanced, optional). We can attempt a more exact phrasing of the PECTT as follows. Suppose that Z is a physical system that accepts n binary stimuli and has a binary output, and can be enclosed in a sphere of volume V. We say that the system Z *computes* a function $f : \{0,1\}^n \rightarrow \{0,1\}$ within t seconds if whenever we set the stimuli to some value $x \in \{0,1\}^n$, if we measure the output after t seconds then we obtain $f(x)$.

One can then phrase the PECTT as stipulating that if there exists such a system Z that computes F within t seconds, then there exists a NAND-CIRC program that computes F and has at most $\alpha(Vt)^2$ lines, where α is some normalization constant. (We can also consider variants where we use surface area instead of volume, or take (Vt) to a different power than 2. However, none of these choices makes a qualitative difference to the discussion below.) In particular, suppose that $f : \{0,1\}^n \rightarrow \{0,1\}$ is a function that requires $2^n/(100n) > 2^{0.8n}$ lines for any NAND-CIRC program (such a function exists by Theorem 5.3). Then the PECTT would imply that either the volume or the time of a system that computes F will have to be at least $2^{0.2n}/\sqrt{\alpha}$. Since this quantity grows *exponentially* in n, it is not hard to set parameters so that even for moderately large values of n, such a system could not fit in our universe.

To fully make the PECTT concrete, we need to decide on the units for measuring time and volume, and the normalization constant α. One conservative choice is to assume that we could squeeze computation to the absolute physical limits (which are many orders of magnitude beyond current technology). This corresponds to setting $\alpha = 1$ and using the Planck units for volume and time. The *Planck length* ℓ_P (which is, roughly speaking, the shortest distance that can theoretically be measured) is roughly 2^{-120} meters. The

Planck time t_P (which is the time it takes for light to travel one Planck length) is about 2^{-150} seconds. In the above setting, if a function F takes, say, 1KB of input (e.g., roughly 10^4 bits, which can encode a 100 by 100 bitmap image), and requires at least $2^{0.8n} = 2^{0.8 \cdot 10^4}$ NAND lines to compute, then any physical system that computes it would require either volume of $2^{0.2 \cdot 10^4}$ Planck length cubed, which is more than 2^{1500} meters cubed or take at least $2^{0.2 \cdot 10^4}$ Planck Time units, which is larger than 2^{1500} seconds. To get a sense of how big that number is, note that the universe is only about 2^{60} seconds old, and its observable radius is only roughly 2^{90} meters. The above discussion suggests that it is possible to *empirically falsify* the PECTT by presenting a smaller-than-universe-size system that computes such a function.

There are of course several hurdles to refuting the PECTT in this way, one of which is that we can't actually test the system on all possible inputs. However, it turns out that we can get around this issue using notions such as *interactive proofs* and *program checking* that we might encounter later in this book. Another, perhaps more salient problem, is that while we know many hard functions exist, at the moment there is *no single explicit function* $F : \{0,1\}^n \rightarrow \{0,1\}$ for which we can *prove* an $\omega(n)$ (let alone $\Omega(2^n/n)$) lower bound on the number of lines that a NAND-CIRC program needs to compute it.

5.6.1 Attempts at refuting the PECTT

One of the admirable traits of mankind is the refusal to accept limitations. In the best case this is manifested by people achieving long-standing "impossible" challenges such as heavier-than-air flight, putting a person on the moon, circumnavigating the globe, or even resolving Fermat's Last Theorem. In the worst case it is manifested by people continually following the footsteps of previous failures to try to do proven-impossible tasks such as build a perpetual motion machine, trisect an angle with a compass and straightedge, or refute Bell's inequality. The Physical Extended Church-Turing thesis (in its various forms) has attracted both types of people. Here are some physical devices that have been speculated to achieve computational tasks that cannot be done by not-too-large NAND-CIRC programs:

- **Spaghetti sort:** One of the first lower bounds that Computer Science students encounter is that sorting n numbers requires making $\Omega(n \log n)$ comparisons. The "spaghetti sort" is a description of a proposed "mechanical computer" that would do this faster. The idea is that to sort n numbers $x_1, \ldots, x_n$, we could cut n spaghetti noodles into lengths $x_1, \ldots, x_n$, and then if we simply hold them

together in our hand and bring them down to a flat surface, they will emerge in sorted order. There are a great many reasons why this is not truly a challenge to the PECTT hypothesis, and I will not ruin the reader's fun in finding them out by her or himself.

- **Soap bubbles:** One function $F : \{0,1\}^n \rightarrow \{0,1\}$ that is conjectured to require a large number of NAND lines to solve is the *Euclidean Steiner Tree* problem. This is the problem where one is given m points in the plane $(x_1, y_1), \ldots, (x_m, y_m)$ (say with integer coordinates ranging from 1 till m, and hence the list can be represented as a string of $n = O(m \log m)$ size) and some number K. The goal is to figure out whether it is possible to connect all the points by line segments of total length at most K. This function is conjectured to be hard because it is *NP complete* - a concept that we'll encounter later in this course - and it is in fact reasonable to conjecture that as m grows, the number of NAND lines required to compute this function grows *exponentially* in m, meaning that the PECTT would predict that if m is sufficiently large (such as few hundreds or so) then no physical device could compute F. Yet, some people claimed that there is in fact a very simple physical device that could solve this problem, that can be constructed using some wooden pegs and soap. The idea is that if we take two glass plates, and put m wooden pegs between them in the locations $(x_1, y_1), \ldots, (x_m, y_m)$ then bubbles will form whose edges touch those pegs in a way that will minimize the total energy which turns out to be a function of the total length of the line segments. The problem with this device is that nature, just like people, often gets stuck in "local optima". That is, the resulting configuration will not be one that achieves the absolute minimum of the total energy but rather one that can't be improved with local changes. Aaronson has carried out actual experiments (see Fig. 5.8), and saw that while this device often is successful for three or four pegs, it starts yielding suboptimal results once the number of pegs grows beyond that.

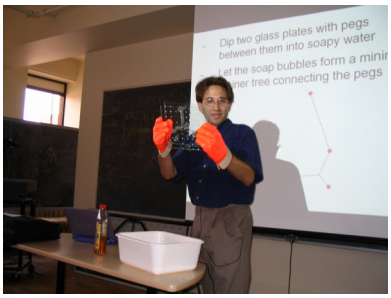


Figure 5.8: Scott Aaronson tests a candidate device for computing Steiner trees using soap bubbles.

- **DNA computing.** People have suggested using the properties of DNA to do hard computational problems. The main advantage of DNA is the ability to potentially encode a lot of information in a relatively small physical space, as well as compute on this information in a highly parallel manner. At the time of this writing, it was demonstrated that one can use DNA to store about 10^{16} bits of information in a region of radius about a millimeter, as opposed to about 10^{10} bits with the best known hard disk technology. This does not posit a real challenge to the PECTT but does suggest that one should be conservative about the choice of constant and not as-

sume that current hard disk + silicon technologies are the absolute best possible.[7]

- **Continuous/real computers.** The physical world is often described using continuous quantities such as time and space, and people have suggested that analog devices might have direct access to computing with real-valued quantities and would be inherently more powerful than discrete models such as NAND machines. Whether the "true" physical world is continuous or discrete is an open question. In fact, we do not even know how to precisely *phrase* this question, let alone answer it. Yet, regardless of the answer, it seems clear that the effort to measure a continuous quantity grows with the level of accuracy desired, and so there is no "free lunch" or way to bypass the PECTT using such machines (see also this paper). Related to that are proposals known as "hypercomputing" or "Zeno's computers" which attempt to use the continuity of time by doing the first operation in one second, the second one in half a second, the third operation in a quarter second and so on.. These fail for a similar reason to the one guaranteeing that Achilles will eventually catch the tortoise despite the original Zeno's paradox.

- **Relativity computer and time travel.** The formulation above assumed the notion of time, but under the theory of relativity time is in the eye of the observer. One approach to solve hard problems is to leave the computer to run for a lot of time from *his* perspective, but to ensure that this is actually a short while from *our* perspective. One approach to do so is for the user to start the computer and then go for a quick jog at close to the speed of light before checking on its status. Depending on how fast one goes, few seconds from the point of view of the user might correspond to centuries in computer time (it might even finish updating its Windows operating system!). Of course the catch here is that the energy required from the user is proportional to how close one needs to get to the speed of light. A more interesting proposal is to use time travel via *closed timelike curves* (*CTCs*). In this case we could run an arbitrarily long computation by doing some calculations, remembering the current state, and then travelling back in time to continue where we left off. Indeed, if CTCs exist then we'd probably have to revise the PECTT (though in this case I will simply travel back in time and edit these notes, so I can claim I never conjectured it in the first place...)

- **Humans.** Another computing system that has been proposed as a counterexample to the PECTT is a 3 pound computer of about 0.1m radius, namely the human brain. Humans can walk around, talk, feel, and do other things that are not commonly done by

[7] We were extremely conservative in the suggested parameters for the PECTT, having assumed that as many as $\ell_P^{-2} 10^{-6} \sim 10^{61}$ bits could potentially be stored in a millimeter radius region.

NAND-CIRC programs, but can they compute partial functions that NAND-CIRC programs cannot? There are certainly computational tasks that *at the moment* humans do better than computers (e.g., play some video games, at the moment), but based on our current understanding of the brain, humans (or other animals) have no *inherent* computational advantage over computers. The brain has about 10^{11} neurons, each operating at a speed of about 1000 operations per seconds. Hence a rough first approximation is that a Boolean circuit of about 10^{14} gates could simulate one second of a brain's activity.[8] Note that the fact that such a circuit (likely) exists does not mean it is easy to *find* it. After all, constructing this circuit took evolution billions of years. Much of the recent efforts in artificial intelligence research is focused on finding programs that replicate some of the brain's capabilities and they take massive computational effort to discover, these programs often turn out to be much smaller than the pessimistic estimates above. For example, at the time of this writing, Google's neural network for machine translation has about 10^4 nodes (and can be simulated by a NAND-CIRC program of comparable size). Philosophers, priests and many others have since time immemorial argued that there is something about humans that cannot be captured by mechanical devices such as computers; whether or not that is the case, the evidence is thin that humans can perform computational tasks that are inherently impossible to achieve by computers of similar complexity.[9]

[8] This is a very rough approximation that could be wrong to a few orders of magnitude in either direction. For one, there are other structures in the brain apart from neurons that one might need to simulate, hence requiring higher overhead. On the other hand, it is by no mean clear that we need to fully clone the brain in order to achieve the same computational tasks that it does.

[9] There are some well known scientists that have advocated that humans have inherent computational advantages over computers. See also this.

- **Quantum computation.** The most compelling attack on the Physical Extended Church-Turing Thesis comes from the notion of *quantum computing*. The idea was initiated by the observation that systems with strong quantum effects are very hard to simulate on a computer. Turning this observation on its head, people have proposed using such systems to perform computations that we do not know how to do otherwise. At the time of this writing, scalable quantum computers have not yet been built, but it is a fascinating possibility, and one that does not seem to contradict any known law of nature. We will discuss quantum computing in much more detail in Chapter 23. Modeling quantum computation involves extending the model of Boolean circuits into *Quantum circuits* that have one more (very special) gate. However, the main takeaway is that while quantum computing does suggest we need to amend the PECTT, it does *not* require a complete revision of our worldview. Indeed, almost all of the content of this book remains the same regardless of whether the underlying computational model is Boolean circuits or quantum circuits.

Remark 5.14 — Physical Extended Church-Turing Thesis and Cryptography. While even the precise phrasing of the PECTT, let alone understanding its correctness, is still a subject of active research, some variants of it are already implicitly assumed in practice. Governments, companies, and individuals currently rely on *cryptography* to protect some of their most precious assets, including state secrets, control of weapon systems and critical infrastructure, securing commerce, and protecting the confidentiality of personal information. In applied cryptography, one often encounters statements such as "cryptosystem X provides 128 bits of security". What such a statement really means is that **(a)** it is conjectured that there is no Boolean circuit (or, equivalently, a NAND-CIRC program) of size much smaller than 2^{128} that can break X, and **(b)** we assume that no other physical mechanism can do better, and hence it would take roughly a 2^{128} amount of "resources" to break X. We say "conjectured" and not "proved" because, while we can phrase the statement that breaking the system cannot be done by an s-gate circuit as a precise mathematical conjecture, at the moment we are unable to *prove* such a statement for any non-trivial cryptosystem. This is related to the **P** vs **NP** question we will discuss in future chapters. We will explore Cryptography in Chapter 21.

Chapter Recap

- We can think of programs both as describing a *process*, as well as simply a list of symbols that can be considered as *data* that can be fed as input to other programs.
- We can write a NAND-CIRC program that evaluates arbitrary NAND-CIRC programs (or equivalently a circuit that evaluates other circuits). Moreover, the efficiency loss in doing so is not too large.
- We can even write a NAND-CIRC program that evaluates programs in other programming languages such as Python, C, Lisp, Java, Go, etc.
- By a leap of faith, we could hypothesize that the number of gates in the smallest circuit that computes a function f captures roughly the amount of physical resources required to compute f. This statement is known as the *Physical Extended Church-Turing Thesis* (*PECTT*).
- Boolean circuits (or equivalently AON-CIRC or NAND-CIRC programs) capture a surprisingly wide array of computational models. The strongest currently known challenge to the PECTT comes from the potential for using quantum mechanical

effects to speed-up computation, a model known as *quantum computers*.

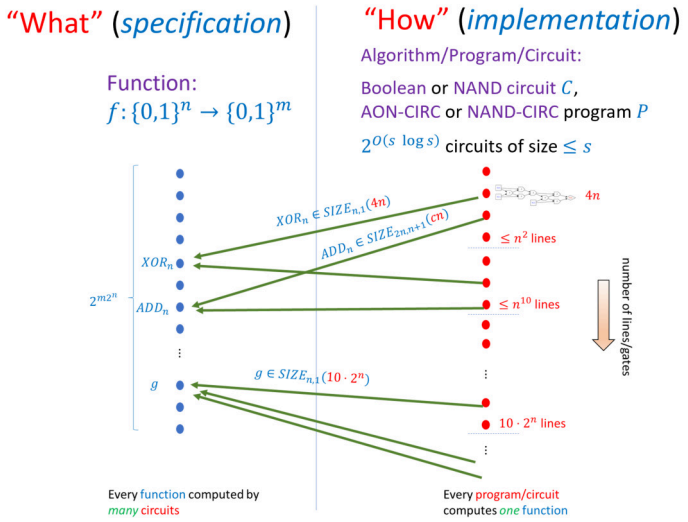


Figure 5.9: A finite computational task is specified by a function $f : \{0,1\}^n \to \{0,1\}^m$. We can model a computational process using Boolean circuits (of varying gate sets) or straight-line program. Every function can be computed by many programs. We say that $f \in SIZE_{n,m}(s)$ if there exists a NAND circuit of at most s gates (equivalently a NAND-CIRC program of at most s lines) that computes f. Every function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a circuit of $O(m \cdot 2^n/n)$ gates. Many functions such as multiplication, addition, solving linear equations, computing the shortest path in a graph, and others, can be computed by circuits of much fewer gates. In particular there is an $O(s^2 \log s)$-size circuit that computes the map $C, x \mapsto C(x)$ where C is a string describing a circuit of s gates. However, the counting argument shows there do exist some functions $f : \{0,1\}^n \to \{0,1\}^m$ that require $\Omega(m \cdot 2^n/n)$ gates to compute.

5.7 RECAP OF PART I: FINITE COMPUTATION

This chapter concludes the first part of this book that deals with *finite computation* (computing functions that map a fixed number of Boolean inputs to a fixed number of Boolean outputs). The main take-aways from Chapter 3, Chapter 4, and Chapter 5 are as follows (see also Fig. 5.9):

- We can formally define the notion of a function $f : \{0,1\}^n \to \{0,1\}^m$ being computable using s basic operations. Whether these operations are AND/OR/NOT, NAND, or some other universal basis does not make much difference. We can describe such a computation either using a *circuit* or using a *straight-line program*.
- We define $SIZE_{n,m}(s)$ to be the set of *functions* that are computable by NAND circuits of at most s gates. This set is equal to the set of functions computable by a NAND-CIRC program of at most s lines and up to a constant factor in s (which we will not care about); this is also the same as the set of functions that are computable by a Boolean circuit of at most s AND/OR/NOT gates. The class $SIZE_{n,m}(s)$ is a set of *functions*, not of programs/circuits.
- *Every* function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed using a circuit of *at most* $O(m \cdot 2^n/n)$ gates. *Some* functions require *at least* $\Omega(m \cdot 2^n/n)$ gates. We define $SIZE_{n,m}(s)$ to be the set of functions from $\{0,1\}^n$ to $\{0,1\}^m$ that can be computed using at most s gates.

- We can describe a circuit/program P as a string. For every s, there is a *universal* circuit/program U_s that can evaluate programs of length s given their description as strings. We can use this representation also to *count* the number of circuits of at most s gates and hence prove that some functions cannot be computed by circuits of smaller-than-exponential size.
- If there is a circuit of s gates that computes a function f, then we can build a physical device to compute f using s basic components (such as transistors). The "Physical Extended Church-Turing Thesis" postulates that the reverse direction is true as well: if f is a function for which *every* circuit requires at least s gates then that *every* physical device to compute f will require about s "physical resources". The main challenge to the PECTT is *quantum computing*, which we will discuss in Chapter 23.

Sneak preview: In the next part we will discuss how to model computational tasks on *unbounded inputs*, which are specified using functions $F : \{0,1\}^* \rightarrow \{0,1\}^*$ (or $F : \{0,1\}^* \rightarrow \{0,1\}$) that can take an unbounded number of Boolean inputs.

5.8 EXERCISES

Exercise 5.1 Which one of the following statements is false:

a. There is an $O(s^3)$ line NAND-CIRC program that given as input program P of s lines in the list-of-tuples representation computes the output of P when all its input are equal to 1.

b. There is an $O(s^3)$ line NAND-CIRC program that given as input program P of s characters encoded as a string of $7s$ bits using the ASCII encoding, computes the output of P when all its input are equal to 1.

c. There is an $O(\sqrt{s})$ line NAND-CIRC program that given as input program P of s lines in the list-of-tuples representation computes the output of P when all its input are equal to 1.

■

Exercise 5.2 — Equals function. For every $k \in \mathbb{N}$, show that there is an $O(k)$ line NAND-CIRC program that computes the function $EQUALS_k : \{0,1\}^{2k} \rightarrow \{0,1\}$ where $EQUALS(x,x') = 1$ if and only if $x = x'$.

■

Exercise 5.3 — Equal to constant function. For every $k \in \mathbb{N}$ and $x' \in \{0,1\}^k$, show that there is an $O(k)$ line NAND-CIRC program that computes the function $EQUALS_{x'} : \{0,1\}^k \rightarrow \{0,1\}$ that on input $x \in \{0,1\}^k$ outputs 1 if and only if $x = x'$.

Exercise 5.4 — Counting lower bound for multibit functions. Prove that there exists a number $\delta > 0$ such that for every n, m there exists a function $f : \{0,1\}^n \to \{0,1\}^m$ that requires at least $\delta m \cdot 2^n/n$ NAND gates to compute. See footnote for hint.[10]

[10] How many functions from $\{0,1\}^n$ to $\{0,1\}^m$ exist?

Exercise 5.5 — Size hierarchy theorem for multibit functions. Prove that there exists a number C such that for every n, m and $n+m < s < m \cdot 2^n/(Cn)$ there exists a function $f \in SIZE_{n,m}(C \cdot s) \setminus SIZE_{n,m}(s)$. See footnote for hint.[11]

[11] Follow the proof of Theorem 5.5, replacing the use of the counting argument with Exercise 5.4.

Exercise 5.6 — Efficient representation of circuits and a tighter counting upper bound. Use the ideas of Remark 5.4 to show that for every $\epsilon > 0$ and sufficiently large s, n, m,

$$|SIZE_{n,m}(s)| < 2^{(2+\epsilon)s \log s + n \log n + m \log s} .$$

Conclude that the implicit constant in Theorem 5.2 can be made arbitrarily close to 5. See footnote for hint.[12]

[12] Using the adjacency list representation, a graph with n in-degree zero vertices and s in-degree two vertices can be represented using roughly $2s \log(s+n) \leq 2s(\log s + O(1))$ bits. The labeling of the n input and m output vertices can be specified by a list of n labels in $[n]$ and m labels in $[m]$.

Exercise 5.7 — Tighter counting lower bound. Prove that for every $\delta < 1/2$, if n is sufficiently large then there exists a function $f : \{0,1\}^n \to \{0,1\}$ such that $f \notin SIZE_{n,1}\left(\frac{\delta 2^n}{n}\right)$. See footnote for hint.[13]

[13] *Hint:* Use the results of Exercise 5.6 and the fact that in this regime $m = 1$ and $n \ll s$.

Exercise 5.8 — Random functions are hard. Suppose $n > 1000$ and that we choose a function $F : \{0,1\}^n \to \{0,1\}$ at random, choosing for every $x \in \{0,1\}^n$ the value $F(x)$ to be the result of tossing an independent unbiased coin. Prove that the probability that there is a $2^n/(1000n)$ line program that computes F is at most 2^{-100}.[14]

[14] **Hint:** An equivalent way to say this is that you need to prove that the set of functions that can be computed using at most $2^n/(1000n)$ lines has fewer than $2^{-100}2^{2^n}$ elements. Can you see why?

Exercise 5.9 The following is a tuple representing a NAND program:
$(3, 1, ((3,2,2), (4,1,1), (5,3,4), (6,2,1), (7,6,6), (8,0,0), (9,7,8), (10,5,0), (11,9,10)))$.

1. Write a table with the eight values $P(000)$, $P(001)$, $P(010)$, $P(011)$, $P(100)$, $P(101)$, $P(110)$, $P(111)$ in this order.
2. Describe what the programs does in words.

Exercise 5.10 — EVAL with XOR. For every sufficiently large n, let $E_n : \{0,1\}^{n^2} \to \{0,1\}$ be the function that takes an n^2-length string that encodes a pair (P, x) where $x \in \{0,1\}^n$ and P is a NAND program of n inputs, a single output, and at most $n^{1.1}$ lines, and returns the output of P on x.[15] That is, $E_n(P, x) = P(x)$.

[15] Note that if n is big enough, then it is easy to represent such a pair using n^2 bits, since we can represent the program using $O(n^{1.1} \log n)$ bits, and we can always pad our representation to have exactly n^2 length.

Prove that for every sufficiently large n, there *does not exist* an XOR circuit C that computes the function E_n, where a XOR circuit has the *XOR* gate as well as the constants 0 and 1 (see Exercise 3.5). That is, prove that there is some constant n_0 such that for every $n > n_0$ and XOR circuit C of n^2 inputs and a single output, there exists a pair (P, x) such that $C(P, x) \neq E_n(P, x)$.

■

Exercise 5.11 — Learning circuits (challenge, optional, assumes more background). (This exercise assumes background in probability theory and/or machine learning that you might not have at this point. Feel free to come back to it at a later point and in particular after going over Chapter 18.) In this exercise we will use our bound on the number of circuits of size s to show that (if we ignore the cost of computation) every such circuit can be *learned* from not too many training samples. Specifically, if we find a size-s circuit that classifies correctly a training set of $O(s \log s)$ samples from some distribution D, then it is guaranteed to do well on the whole distribution D. Since Boolean circuits model very many physical processes (maybe even all of them, if the (controversial) physical extended Church-Turing thesis is true), this shows that all such processes could be learned as well (again, ignoring the computation cost of finding a classifier that does well on the training data).

Let D be any probability distribution over $\{0,1\}^n$ and let C be a NAND circuit with n inputs, one output, and size $s \geq n$. Prove that there is some constant c such that with probability at least 0.999 the following holds: if $m = cs \log s$ and $x_0, \ldots, x_{m-1}$ are chosen independently from D, then for every circuit C' such that $C'(x_i) = C(x_i)$ on every $i \in [m]$, $\Pr_{x \sim D}[C'(x) \leq C(x)] \leq 0.99$.

In other words, if C' is a so called "empirical risk minimizer" that agrees with C on all the training examples $x_0, \ldots, x_{n-1}$, then it will also agree with C with high probability for samples drawn from the distribution D (i.e., it "generalizes", to use Machine-Learning lingo). See footnote for hint.[16]

■

[16] *Hint:* Use our bound on the number of programs/circuits of size s (Theorem 5.2), as well as the Chernoff Bound (Theorem 18.12) and the union bound.

5.9 BIBLIOGRAPHICAL NOTES

The *EVAL* function is usually known as a *universal circuit*. The implementation we describe is not the most efficient known. Valiant [Val76] first showed a universal circuit of size $O(n \log n)$ where n is the size of the input. Universal circuits have seen in recent years new motivations due to their applications for cryptography, see [LMS16; GKS17] .

While we've seen that "most" functions mapping n bits to one bit require circuits of exponential size $\Omega(2^n/n)$, we actually do not know

of any *explicit* function for which we can *prove* that it requires, say, at least n^{100} or even $100n$ size. At the moment, the strongest such lower bound we know is that there are quite simple and explicit n-variable functions that require at least $(5 - o(1))n$ lines to compute, see this paper of Iwama et al as well as this more recent work of Kulikov et al. Proving lower bounds for restricted models of circuits is an extremely interesting research area, for which Jukna's book [Juk12] (see also Wegener [Weg87]) provides a very good introduction and overview. I learned of the proof of the size hierarchy theorem (Theorem 5.5) from Sasha Golovnev.

Scott Aaronson's blog post on how information is physical is a good discussion on issues related to the physical extended Church-Turing Physics. Aaronson's survey on NP complete problems and physical reality [Aar05] discusses these issues as well, though it might be easier to read after we reach Chapter 15 on **NP** and **NP**-completeness.

II

UNIFORM COMPUTATION

Learning Objectives:

- Define functions on *unbounded length* inputs, that cannot be described by a finite size table of inputs and outputs.
- Equivalence with the task of deciding membership in a language.
- Deterministic finite automatons (optional): A simple example for a model for unbounded computation.
- Equivalence with regular expressions.

6

Functions with Infinite domains, Automata, and Regular expressions

"An algorithm is a finite answer to an infinite number of questions.", Attributed to Stephen Kleene.

The model of Boolean circuits (or equivalently, the NAND-CIRC programming language) has one very significant drawback: a Boolean circuit can only compute a *finite* function f. In particular, since every gate has two inputs, a size s circuit can compute on an input of length at most $2s$. Thus this model does not capture our intuitive notion of an algorithm as a *single recipe* to compute a potentially infinite function. For example, the standard elementary school multiplication algorithm is a *single* algorithm that multiplies numbers of all lengths. However, we cannot express this algorithm as a single circuit, but rather need a different circuit (or equivalently, a NAND-CIRC program) for every input length (see Fig. 6.1).

In this chapter, we extend our definition of computational tasks to consider functions with the *unbounded* domain of $\{0,1\}^*$. We focus on the question of defining **what** tasks to compute, mostly leaving the question of **how** to compute them to later chapters, where we will see *Turing machines* and other computational models for computing on unbounded inputs. However, we will see one example of a simple restricted model of computation - deterministic finite automata (DFAs).

This chapter: A non-mathy overview

In this chapter, we discuss functions that take as input strings of arbitrary length. We will often focus on the special case of *Boolean* functions, where the output is a single bit. These are still infinite functions since their inputs have unbounded

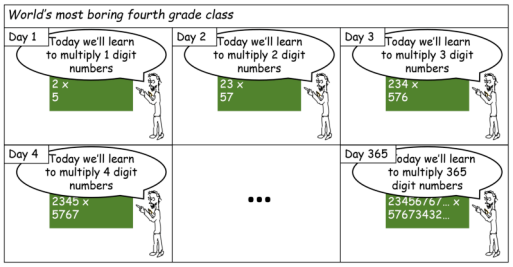


Figure 6.1: Once you know how to multiply multi-digit numbers, you can do so for every number n of digits, but if you had to describe multiplication using Boolean circuits or NAND-CIRC programs, you would need a different program/circuit for every length n of the input.

length and hence such a function cannot be computed by any single Boolean circuit.
In the second half of this chapter, we discuss *finite automata*, a computational model that can compute unbounded length functions. Finite automata are not as powerful as Python or other general-purpose programming languages but can serve as an introduction to these more general models. We also show a beautiful result - the functions computable by finite automata are precisely the ones that correspond to *regular expressions*. However, the reader can also feel free to skip automata and go straight to our discussion of *Turing machines* in Chapter 7.

6.1 FUNCTIONS WITH INPUTS OF UNBOUNDED LENGTH

Up until now, we considered the computational task of mapping some string of length n into a string of length m. However, in general, computational tasks can involve inputs of *unbounded* length. For example, the following Python function computes the function $XOR : \{0,1\}^* \rightarrow \{0,1\}$, where $XOR(x)$ equals 1 iff the number of 1's in x is odd. (In other words, $XOR(x) = \sum_{i=0}^{|x|-1} x_i \mod 2$ for every $x \in \{0,1\}^*$.) As simple as it is, the *XOR* function cannot be computed by a Boolean circuit. Rather, for every n, we can compute XOR_n (the restriction of *XOR* to $\{0,1\}^n$) using a different circuit (e.g., see Fig. 6.2).

```
def XOR(X):
    '''Takes list X of 0's and 1's
       Outputs 1 if the number of 1's is odd and outputs 0
    otherwise'''
    result = 0
    for i in range(len(X)):
        result = (result + X[i]) % 2
    return result
```

Previously in this book, we studied the computation of *finite* functions $f : \{0,1\}^n \rightarrow \{0,1\}^m$. Such a function f can always be described by listing all the 2^n values it takes on inputs $x \in \{0,1\}^n$. In this chapter, we consider functions such as *XOR* that take inputs of *unbounded* size. While we can describe *XOR* using a finite number of symbols (in fact, we just did so above), it takes infinitely many possible inputs, and so we cannot just write down all of its values. The same is true for many other functions capturing important computational tasks, including addition, multiplication, sorting, finding paths in

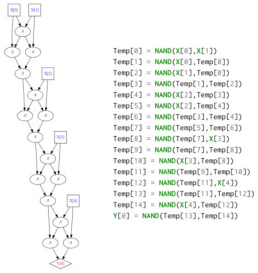


Figure 6.2: The NAND circuit and NAND-CIRC program for computing the XOR of 5 bits. Note how the circuit for XOR_5 merely repeats four times the circuit to compute the XOR of 2 bits.

graphs, fitting curves to points, and so on. To contrast with the finite case, we will sometimes call a function $F : \{0,1\}^* \rightarrow \{0,1\}$ (or $F : \{0,1\}^* \rightarrow \{0,1\}^*$) *infinite*. However, this does not mean that F takes as input strings of infinite length! It just means that F can take as input a string of that can be arbitrarily long, and so we cannot simply write down a table of all the outputs of F on different inputs.

Big Idea 8 A function $F : \{0,1\}^* \rightarrow \{0,1\}^*$ specifies the computational task mapping an input $x \in \{0,1\}^*$ into the output $F(x)$.

As we have seen before, restricting attention to functions that use binary strings as inputs and outputs does not detract from our generality, since other objects, including numbers, lists, matrices, images, videos, and more, can be encoded as binary strings.

As before, it is essential to differentiate between *specification* and *implementation*. For example, consider the following function:

$$TWINP(x) = \begin{cases} 1 & \exists_{p \in \mathbb{N}} \text{ s.t.} p, p+2 \text{ are primes and } p > |x| \\ 0 & \text{otherwise} \end{cases}$$

This is a mathematically well-defined function. For every x, $TWINP(x)$ has a unique value which is either 0 or 1. However, at the moment, no one knows of a *Python* program that computes this function. The Twin prime conjecture posits that for every n there exists $p > n$ such that both p and $p+2$ are primes. If this conjecture is true, then T is easy to compute indeed - the program `def T(x): return 1` will do the trick. However, mathematicians have tried unsuccessfully to prove this conjecture since 1849. That said, whether or not we know how to *implement* the function *TWINP*, the definition above provides its *specification*.

6.1.1 Varying inputs and outputs

Many of the functions that interest us take more than one input. For example, the function

$$MULT(x,y) = x \cdot y$$

takes the binary representation of a pair of integers $x, y \in \mathbb{N}$, and outputs the binary representation of their product $x \cdot y$. However, since we can represent a pair of strings as a single string, we will consider functions such as MULT as mapping $\{0,1\}^*$ to $\{0,1\}^*$. We will typically not be concerned with low-level details such as the precise way to represent a pair of integers as a string, since virtually all choices will be equivalent for our purposes.

Another example of a function we want to compute is

$$PALINDROME(x) = \begin{cases} 1 & \forall_{i \in [|x|]} x_i = x_{|x|-i} \\ 0 & \text{otherwise} \end{cases}$$

PALINDROME has a single bit as output. Functions with a single bit of output are known as *Boolean functions*. Boolean functions are central to the theory of computation, and we will discuss them often in this book. Note that even though Boolean functions have a single bit of output, their *input* can be of arbitrary length. Thus they are still infinite functions that cannot be described via a finite table of values.

"Booleanizing" functions. Sometimes it might be convenient to obtain a Boolean variant for a non-Boolean function. For example, the following is a Boolean variant of *MULT*.

$$BMULT(x, y, i) = \begin{cases} i^{th} \text{ bit of } x \cdot y & i < |x \cdot y| \\ 0 & \text{otherwise} \end{cases}$$

If we can compute *BMULT* via any programming language such as Python, C, Java, etc., we can compute *MULT* as well, and vice versa.

Solved Exercise 6.1 — Booleanizing general functions. Show that for every function $F : \{0,1\}^* \to \{0,1\}^*$, there exists a Boolean function $BF : \{0,1\}^* \to \{0,1\}$ such that a Python program to compute BF can be transformed into a program to compute F and vice versa.

■

Solution:

For every $F : \{0,1\}^* \to \{0,1\}^*$, we can define

$$BF(x, i, b) = \begin{cases} F(x)_i & i < |F(x)|, b = 0 \\ 1 & i < |F(x)|, b = 1 \\ 0 & i \geq |F(x)| \end{cases}$$

to be the function that on input $x \in \{0,1\}^*, i \in \mathbb{N}, b \in \{0,1\}$ outputs the ith bit of $F(x)$ if $b = 0$ and $i < |F(x)|$. If $b = 1$, then $BF(x, i, b)$ outputs 1 iff $i < |F(x)|$ and hence this allows to compute the length of $F(x)$.

Computing BF from F is straightforward. For the other direction, given a Python function `BF` that computes BF, we can compute F as follows:

```
def F(x):
    res = []
    i = 0
    while BF(x,i,1):
```

```
        res.append(BF(x,i,0))
        i += 1
    return res
```

■

6.1.2 Formal Languages

For every Boolean function $F : \{0,1\}^* \to \{0,1\}$, we can define the set $L_F = \{x | F(x) = 1\}$ of strings on which F outputs 1. Such sets are known as *languages*. This name is rooted in *formal language theory* as pursued by linguists such as Noam Chomsky. A *formal language* is a subset $L \subseteq \{0,1\}^*$ (or more generally $L \subseteq \Sigma^*$ for some finite alphabet Σ). The *membership* or *decision* problem for a language L, is the task of determining, given $x \in \{0,1\}^*$, whether or not $x \in L$. If we can compute the function F, then we can decide membership in the language L_F and vice versa. Hence, many texts such as [Sip97] refer to the task of computing a Boolean function as "deciding a language". In this book, we mostly describe computational tasks using the *function* notation, which is easier to generalize to computation with more than one bit of output. However, since the language terminology is so popular in the literature, we will sometimes mention it.

6.1.3 Restrictions of functions

If $F : \{0,1\}^* \to \{0,1\}$ is a Boolean function and $n \in \mathbb{N}$ then the *restriction* of F to inputs of length n, denoted as F_n, is the finite function $f : \{0,1\}^n \to \{0,1\}$ such that $f(x) = F(x)$ for every $x \in \{0,1\}^n$. That is, F_n is the finite function that is only defined on inputs in $\{0,1\}^n$, but agrees with F on those inputs. Since F_n is a finite function, it can be computed by a Boolean circuit, implying the following theorem:

Theorem 6.1 — Circuit collection for every infinite function. Let $F : \{0,1\}^* \to \{0,1\}$. Then there is a collection $\{C_n\}_{n \in \{1,2,\ldots\}}$ of circuits such that for every $n > 0$, C_n computes the restriction F_n of F to inputs of length n.

Proof. This is an immediate corollary of the universality of Boolean circuits. Indeed, since F_n maps $\{0,1\}^n$ to $\{0,1\}$, Theorem 4.15 implies that there exists a Boolean circuit C_n to compute it. In fact, the size of this circuit is at most $c \cdot 2^n/n$ gates for some constant $c \leq 10$.

■

In particular, Theorem 6.1 implies that there exists such a circuit collection $\{C_n\}$ even for the *TWINP* function we described before, even though we do not know of any program to compute it. Indeed, this is not that surprising: for every particular $n \in \mathbb{N}$, $TWINP_n$ is either the constant zero function or the constant one function, both of which

can be computed by very simple Boolean circuits. Hence a collection of circuits $\{C_n\}$ that computes *TWINP* certainly exists. The difficulty in computing *TWINP* using Python or any other programming language arises from the fact that we do not know for each particular n what is the circuit C_n in this collection.

6.2 DETERMINISTIC FINITE AUTOMATA (OPTIONAL)

All our computational models so far - Boolean circuits and straight-line programs - were only applicable for *finite* functions.

In Chapter 7, we will present *Turing machines*, which are the central models of computation for unbounded input length functions. However, in this section we present the more basic model of *deterministic finite automata* (DFA). Automata can serve as a good stepping-stone for Turing machines, though they will not be used much in later parts of this book, and so the reader can feel free to skip ahead to Chapter 7. DFAs turn out to be equivalent in power to *regular expressions*: a powerful mechanism to specify patterns, which is widely used in practice. Our treatment of automata is relatively brief. There are plenty of resources that help you get more comfortable with DFAs. In particular, Chapter 1 of Sipser's book [Sip97] contains an excellent exposition of this material. There are also many websites with online simulators for automata, as well as translators from regular expressions to automata and vice versa (see for example here and here).

At a high level, an *algorithm* is a recipe for computing an output from an input via a combination of the following steps:

1. Read a bit from the input
2. Update the *state* (working memory)
3. Stop and produce an output

For example, recall the Python program that computes the *XOR* function:

```
def XOR(X):
    '''Takes list X of 0's and 1's
       Outputs 1 if the number of 1's is odd and outputs 0
    otherwise'''
    result = 0
    for i in range(len(X)):
        result = (result + X[i]) % 2
    return result
```

In each step, this program reads a single bit `X[i]` and updates its state `result` based on that bit (flipping `result` if `X[i]` is 1 and keeping it the same otherwise). When it is done transversing the input,

the program outputs `result`. In computer science, such a program is called a *single-pass constant-memory algorithm* since it makes a single pass over the input and its working memory is finite. (Indeed, in this case, `result` can either be 0 or 1.) Such an algorithm is also known as a *Deterministic Finite Automaton* or *DFA* (another name for DFAs is a *finite state machine*). We can think of such an algorithm as a "machine" that can be in one of C states, for some constant C. The machine starts in some initial state and then reads its input $x \in \{0,1\}^*$ one bit at a time. Whenever the machine reads a bit $\sigma \in \{0,1\}$, it transitions into a new state based on σ and its prior state. The output of the machine is based on the final state. Every single-pass constant-memory algorithm corresponds to such a machine. If an algorithm uses c bits of memory, then the contents of its memory can be represented as a string of length c. Therefore such an algorithm can be in one of at most 2^c states at any point in the execution.

We can specify a DFA of C states by a list of $C \cdot 2$ rules. Each rule will be of the form "If the DFA is in state v and the bit read from the input is σ then the new state is v'". At the end of the computation, we will also have a rule of the form "If the final state is one of the following ... then output 1, otherwise output 0". For example, the Python program above can be represented by a two-state automaton for computing *XOR* of the following form:

- Initialize in the state 0.
- For every state $s \in \{0,1\}$ and input bit σ read, if $\sigma = 1$ then change to state $1 - s$, otherwise stay in state s.
- At the end output 1 iff $s = 1$.

We can also describe a C-state DFA as a labeled *graph* of C vertices. For every state s and bit σ, we add a directed edge labeled with σ between s and the state s' such that if the DFA is at state s and reads σ then it transitions to s'. (If the state stays the same then this edge will be a self-loop; similarly, if s transitions to s' in both the case $\sigma = 0$ and $\sigma = 1$ then the graph will contain two parallel edges.) We also label the set $\mathcal{S}$ of states on which the automaton will output 1 at the end of the computation. This set is known as the set of *accepting states*. See Fig. 6.3 for the graphical representation of the XOR automaton.

Formally, a DFA is specified by **(1)** the table of the $C \cdot 2$ rules, which can be represented as a *transition function* T that maps a state $s \in [C]$ and bit $\sigma \in \{0,1\}$ to the state $s' \in [C]$ which the DFA will transition to from state s on input σ and **(2)** the set $\mathcal{S}$ of accepting states. This leads to the following definition.

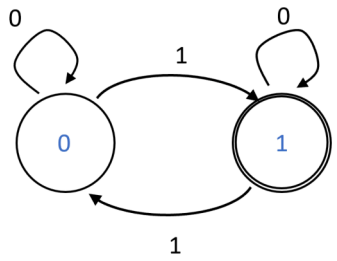


Figure 6.3: A deterministic finite automaton that computes the *XOR* function. It has two states 0 and 1, and when it observes σ it transitions from v to $v \oplus \sigma$.

Definition 6.2 — Deterministic Finite Automaton. A deterministic finite automaton (DFA) with C states over $\{0,1\}$ is a pair $(T, \mathcal{S})$ with $T : [C] \times \{0,1\} \to [C]$ and $\mathcal{S} \subseteq [C]$. The finite function T is known as the *transition function* of the DFA. The set $\mathcal{S}$ is known as the set of *accepting states*.

Let $F : \{0,1\}^* \to \{0,1\}$ be a Boolean function with the infinite domain $\{0,1\}^*$. We say that $(T, \mathcal{S})$ *computes* a function $F : \{0,1\}^* \to \{0,1\}$ if for every $n \in \mathbb{N}$ and $x \in \{0,1\}^n$, if we define $s_0 = 0$ and $s_{i+1} = T(s_i, x_i)$ for every $i \in [n]$, then

$$s_n \in \mathcal{S} \Leftrightarrow F(x) = 1$$

P

Make sure not to confuse the *transition function* of an automaton (T in Definition 6.2), which is a finite function specifying the table of "rules" which it follows, with the function the automaton *computes* (F in Definition 6.2) which is an infinite function.

R

Remark 6.3 — Definitions in other texts. Deterministic finite automata can be defined in several equivalent ways. In particular Sipser [Sip97] defines a DFA as a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, Σ is the alphabet, δ is the transition function, q_0 is the initial state, and F is the set of accepting states. In this book the set of states is always of the form $Q = \{0, \ldots, C-1\}$ and the initial state is always $q_0 = 0$, but this makes no difference to the computational power of these models. Also, we restrict our attention to the case that the alphabet Σ is equal to $\{0,1\}$.

Solved Exercise 6.2 — DFA for $(010)^*$. Prove that there is a DFA that computes the following function F:

$$F(x) = \begin{cases} 1 & 3 \text{ divides } |x| \text{ and } \forall_{i \in [|x|/3]} x_{3i} x_{3i+1} x_{3i+2} = 010 \\ 0 & \text{otherwise} \end{cases}$$

■

Solution:

When asked to construct a deterministic finite automaton, it is often useful to start by constructing a single-pass constant-memory

algorithm using a more general formalism (for example, using pseudocode or a *Python* program). Once we have such an algorithm, we can mechanically translate it into a DFA. Here is a simple Python program for computing F:

```
def F(X):
    '''Return 1 iff X is a concatenation of zero/more
     ↪  copies of [0,1,0]'''
    if len(X) % 3 != 0:
        return False
    ultimate = 0
    penultimate = 1
    antepenultimate = 0
    for idx, b in enumerate(X):
        antepenultimate = penultimate
        penultimate = ultimate
        ultimate = b
        if idx % 3 == 2 and ((antepenultimate,
         ↪  penultimate, ultimate) != (0,1,0)):
            return False
    return True
```

Since we keep three Boolean variables, the working memory can be in one of $2^3 = 8$ configurations, and so the program above can be directly translated into an 8 state DFA. While this is not needed to solve the question, by examining the resulting DFA, we can see that we can merge some states and obtain a 4 state automaton, described in Fig. 6.4. See also Fig. 6.5, which depicts the execution of this DFA on a particular input.

6.2.1 Anatomy of an automaton (finite vs. unbounded)

Now that we are considering computational tasks with unbounded input sizes, it is crucial to distinguish between the components of our algorithm that have *fixed length* and the components that grow with the input size. For the case of DFAs these are the following:

Constant size components: Given a DFA A, the following quantities are fixed independent of the input size:

- The number of *states* C in A.
- The *transition function* T (which has $2C$ inputs, and so can be specified by a table of $2C$ rows, each entry in which is a number in $[C]$).
- The set $\mathcal{S} \subseteq [C]$ of accepting states. This set can be described by a string in $\{0,1\}^C$ specifiying which states are in $\mathcal{S}$ and which are not.

s	σ	$T(s,\sigma)$
0	0	2
0	1	3
1	0	0
1	1	3
2	0	3
2	1	1
3	0	3
3	1	3

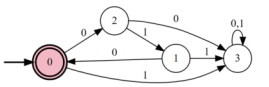


Figure 6.4: A DFA that outputs 1 only on inputs $x \in \{0,1\}^*$ that are a concatenation of zero or more copies of 010. The state 0 is both the starting state and the only accepting state. The table denotes the transition function of T, which maps the current state and symbol read to the new symbol.

Together the above means that we can fully describe an automaton using finitely many symbols. This is a property we require out of any notion of "algorithm": we should be able to write down a complete specification of how it produces an output from an input.

Components of unbounded size: The following quantities relating to a DFA are not bounded by any constant. We stress that these are still *finite* for any given input.

- The length of the input $x \in \{0,1\}^*$ that the DFA is provided. The input length is always finite, but not a priori bounded.
- The number of steps that the DFA takes can grow with the length of the input. Indeed, a DFA makes a single pass on the input and so it takes precisely $|x|$ steps on an input $x \in \{0,1\}^*$.

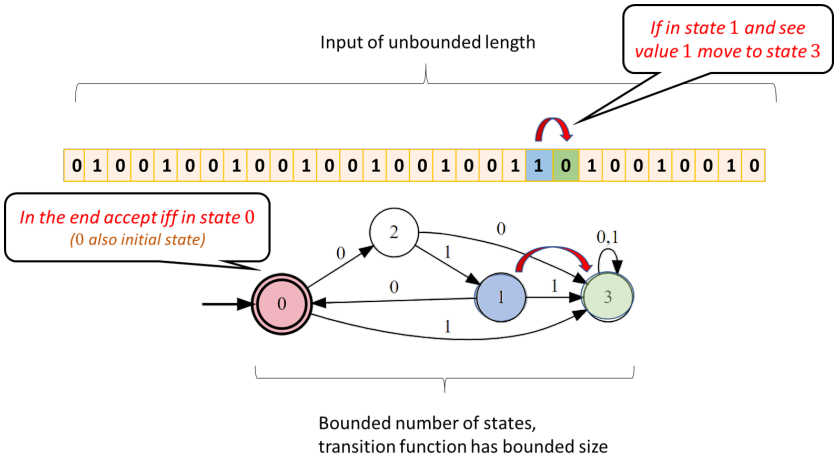


Figure 6.5: Execution of the DFA of Fig. 6.4. The number of states and the transition function size are bounded, but the input can be arbitrarily long. If the DFA is at state s and observes the value σ then it moves to the state $T(s,\sigma)$. At the end of the execution the DFA accepts iff the final state is in $\mathcal{S}$.

6.2.2 DFA-computable functions

We say that a function $F : \{0,1\}^* \to \{0,1\}$ is *DFA computable* if there exists some *DFA* that computes F. In Chapter 4 we saw that every finite function is computable by some Boolean circuit. Thus, at this point, you might expect that every infinite function is computable by *some* DFA. However, this is very much *not* the case. We will soon see some simple examples of infinite functions that are not computable by DFAs, but for starters, let us prove that such functions exist.

Theorem 6.4 — DFA-computable functions are countable. Let *DFACOMP* be the set of all Boolean functions $F : \{0,1\}^* \to \{0,1\}$ such that there exists a DFA computing F. Then *DFACOMP* is countable.

Proof Idea:

Every DFA can be described by a finite length string, which yields an onto map from $\{0,1\}^*$ to *DFACOMP*: namely, the function that maps a string describing an automaton A to the function that it computes.

⋆

Proof of Theorem 6.4. Every DFA can be described by a finite string, representing the transition function T and the set of accepting states, and every DFA A computes *some* function $F : \{0,1\}^* \to \{0,1\}$. Thus we can define the following function $StDC : \{0,1\}^* \to DFACOMP$:

$$StDC(a) = \begin{cases} F & a \text{ represents automaton } A \text{ and } F \text{ is the function } A \text{ computes} \\ ONE & \text{otherwise} \end{cases}$$

where $ONE : \{0,1\}^* \to \{0,1\}$ is the constant function that outputs 1 on all inputs (and is a member of *DFACOMP*). Since by definition, every function F in *DFACOMP* is computable by *some* automaton, $StDC$ is an onto function from $\{0,1\}^*$ to *DFACOMP*, which means that *DFACOMP* is countable (see Section 2.4.2).

■

Since the set of *all* Boolean functions is uncountable, we get the following corollary:

Theorem 6.5 — Existence of DFA-uncomputable functions. There exists a Boolean function $F : \{0,1\}^* \to \{0,1\}$ that is not computable by *any* DFA.

Proof. If every Boolean function F is computable by some DFA, then *DFACOMP* equals the set *ALL* of all Boolean functions, but by Theorem 2.12, the latter set is uncountable, contradicting Theorem 6.4.

■

6.3 REGULAR EXPRESSIONS

Searching for a piece of text is a common task in computing. At its heart, the *search problem* is quite simple. We have a collection $X = \{x_0, \ldots, x_k\}$ of strings (e.g., files on a hard-drive, or student records in a database), and the user wants to find out the subset of all the $x \in X$ that are *matched* by some pattern (e.g., all files whose names end with the string `.txt`). In full generality, we can allow the user to specify the pattern by specifying a (computable) *function* $F : \{0,1\}^* \to \{0,1\}$, where $F(x) = 1$ corresponds to the pattern matching x. That is, the user provides a *program* P in a programming language such as *Python*, and the system returns all $x \in X$ such that $P(x) = 1$. For example,

one could search for all text files that contain the string `important document` or perhaps (letting P correspond to a neural-network based classifier) all images that contain a cat. However, we don't want our system to get into an infinite loop just trying to evaluate the program P! For this reason, typical systems for searching files or databases do *not* allow users to specify the patterns using full-fledged programming languages. Rather, such systems use *restricted computational models* that on the one hand are *rich enough* to capture many of the queries needed in practice (e.g., all filenames ending with `.txt`, or all phone numbers of the form `(617) xxx-xxxx`), but on the other hand are *restricted* enough so that queries can be evaluated very efficiently on huge files and in particular cannot result in an infinite loop.

One of the most popular such computational models is regular expressions. If you ever used an advanced text editor, a command-line shell, or have done any kind of manipulation of text files, then you have probably come across regular expressions.

A *regular expression* over some alphabet Σ is obtained by combining elements of Σ with the operation of concatenation, as well as $|$ (corresponding to *or*) and $*$ (corresponding to repetition zero or more times). (Common implementations of regular expressions in programming languages and shells typically include some extra operations on top of $|$ and $*$, but these operations can be implemented as "syntactic sugar" using the operators $|$ and $*$.) For example, the following regular expression over the alphabet $\{0,1\}$ corresponds to the set of all strings $x \in \{0,1\}^*$ where every digit is repeated at least twice:

$$(00(0^*)|11(1^*))^* \ .$$

The following regular expression over the alphabet $\{a,\ldots,z,0,\ldots,9\}$ corresponds to the set of all strings that consist of a sequence of one or more of the letters a-d followed by a sequence of one or more digits (without a leading zero):

$$(a|b|c|d)(a|b|c|d)^*(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^* \ . \tag{6.1}$$

Formally, regular expressions are defined by the following recursive definition:

Definition 6.6 — Regular expression. A *regular expression* e over an alphabet Σ is a string over $\Sigma \cup \{(,),|,*,\emptyset,\texttt{""}\}$ that has one of the following forms:

1. $e = \sigma$ where $\sigma \in \Sigma$
2. $e = (e'|e'')$ where e', e'' are regular expressions.

3. $e = (e')(e'')$ where e', e'' are regular expressions. (We often drop the parentheses when there is no danger of confusion and so write this as $e'\ e''$.)

4. $e = (e')^*$ where e' is a regular expression.

Finally we also allow the following "edge cases": $e = \emptyset$ and $e =$ "". These are the regular expressions corresponding to accepting no strings, and accepting only the empty string respectively.

We will drop parentheses when they can be inferred from the context. We also use the convention that OR and concatenation are left-associative, and we give highest precedence to $*$, then concatenation, and then OR. Thus for example we write $00^*|11$ instead of $((0)(0^*))|((1)(1))$.

Every regular expression e corresponds to a function $\Phi_e : \Sigma^* \to \{0,1\}$ where $\Phi_e(x) = 1$ if x *matches* the regular expression. For example, if $e = (00|11)^*$ then $\Phi_e(110011) = 1$ but $\Phi_e(101) = 0$ (can you see why?).

The formal definition of Φ_e is one of those definitions that is more cumbersome to write than to grasp. Thus it might be easier for you first to work out the definition on your own, and then check that it matches what is written below.

Definition 6.7 — Matching a regular expression. Let e be a regular expression over the alphabet Σ. The function $\Phi_e : \Sigma^* \to \{0,1\}$ is defined as follows:

1. If $e = \sigma$ then $\Phi_e(x) = 1$ iff $x = \sigma$.

2. If $e = (e'|e'')$ then $\Phi_e(x) = \Phi_{e'}(x) \vee \Phi_{e''}(x)$ where $\vee$ is the OR operator.

3. If $e = (e')(e'')$ then $\Phi_e(x) = 1$ iff there is some $x', x'' \in \Sigma^*$ such that x is the concatenation of x' and x'' and $\Phi_{e'}(x') = \Phi_{e''}(x'') = 1$.

4. If $e = (e')*$ then $\Phi_e(x) = 1$ iff there is some $k \in \mathbb{N}$ and some $x_0, \ldots, x_{k-1} \in \Sigma^*$ such that x is the concatenation $x_0 \cdots x_{k-1}$ and $\Phi_{e'}(x_i) = 1$ for every $i \in [k]$.

5. Finally, for the edge cases $\Phi_\emptyset$ is the constant zero function, and $\Phi_{\text{""}}$ is the function that only outputs 1 on the empty string "".

We say that a regular expression e over Σ *matches* a string $x \in \Sigma^*$ if $\Phi_e(x) = 1$.

The definitions above are not inherently difficult but are a bit cumbersome. So you should pause here and go over it again until you understand why it corresponds to our intuitive notion of regular expressions. This is important not just for understanding regular expressions themselves (which are used time and again in a great many applications) but also for getting better at understanding recursive definitions in general.

A Boolean function is called "regular" if it outputs 1 on precisely the set of strings that are matched by some regular expression. That is,

Definition 6.8 — Regular functions / languages. Let Σ be a finite set and $F : \Sigma^* \to \{0,1\}$ be a Boolean function. We say that F is *regular* if $F = \Phi_e$ for some regular expression e.

Similarly, for every formal language $L \subseteq \Sigma^*$, we say that L is regular if and only if there is a regular expression e such that $x \in L$ iff e matches x.

■ **Example 6.9 — A regular function.** Let $\Sigma = \{a, b, c, d, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $F : \Sigma^* \to \{0,1\}$ be the function such that $F(x)$ outputs 1 iff x consists of one or more of the letters a-d followed by a sequence of one or more digits (without a leading zero). Then F is a regular function, since $F = \Phi_e$ where

$$e = (a|b|c|d)(a|b|c|d)^*(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

is the expression we saw in (6.1).

If we wanted to verify, for example, that $\Phi_e(abc12078) = 1$, we can do so by noticing that the expression $(a|b|c|d)$ matches the string a, $(a|b|c|d)^*$ matches bc, $(0|1|2|3|4|5|6|7|8|9)$ matches the string 1, and the expression $(0|1|2|3|4|5|6|7|8|9)^*$ matches the string 2078. Each one of those boils down to a simpler expression. For example, the expression $(a|b|c|d)^*$ matches the string bc because both of the one-character strings b and c are matched by the expression $a|b|c|d$.

Regular expression can be defined over any finite alphabet Σ, but as usual, we will mostly focus our attention on the *binary case*, where $\Sigma = \{0,1\}$. Most (if not all) of the theoretical and practical general

insights about regular expressions can be gleaned from studying the binary case.

6.3.1 Algorithms for matching regular expressions

Regular expressions would not be very useful for search if we could not evaluate, given a regular expression e, whether a string x is matched by e. Luckily, there is an algorithm to do so. Specifically, there is an algorithm (think "Python program" though later we will formalize the notion of algorithms using *Turing machines*) that on input a regular expression e over the alphabet $\{0,1\}$ and a string $x \in \{0,1\}^*$, outputs 1 iff e matches x (i.e., outputs $\Phi_e(x)$).

Indeed, Definition 6.7 actually specifies a recursive algorithm for *computing* Φ_e. Specifically, each one of our operations -concatenation, OR, and star- can be thought of as reducing the task of testing whether an expression e matches a string x to testing whether some sub-expressions of e match substrings of x. Since these sub-expressions are always shorter than the original expression, this yields a recursive algorithm for checking if e matches x, which will eventually terminate at the base cases of the expressions that correspond to a single symbol or the empty string.

Algorithm 6.10 — Regular expression matching.

Input: Regular expression e over Σ^*, $x \in \Sigma^*$
Output: $\Phi_e(x)$

```
1:  procedure MATCH(e,x)
2:      if e = ∅ then return 0 ;
3:      if x = "" then return MATCHEMPTY(e) ;
4:      if e ∈ Σ then return 1 iff x = e ;
5:      if e = (e'|e'') then return MATCH(e', x) or MATCH(e'', x)
        ;
6:      if e = (e')(e'') then
7:          for i ∈ [|x|] do
8:              if MATCH(e', x_0 ··· x_{i-1}) and MATCH(e'', x_i ··· x_{|x|-1})
        then return 1 ;
9:          end for
10:     end if
11:     if e = (e')* then
12:         if e' = "" then return MATCH("", x) ;
13:                                  # ("")* is the same as ""
14:         for i ∈ [|x|] do
15:                                  # x_0 ··· x_{i-1} is shorter than x
16:             if MATCH(e, x_0 ··· x_{i-1}) and MATCH(e', x_i ··· x_{|x|-1})
        then return 1 ;
17:         end for
18:     end if
19:     return 0
20: end procedure
```

We assume above that we have a procedure MATCHEMPTY that on input a regular expression e outputs 1 if and only if e matches the empty string "".

The key observation is that in our recursive definition of regular expressions, whenever e is made up of one or two expressions e', e'' then these two regular expressions are *smaller* than e. Eventually (when they have size 1) then they must correspond to the non-recursive case of a single alphabet symbol. Correspondingly, the recursive calls made in Algorithm 6.10 always correspond to a shorter expression or (in the case of an expression of the form $(e')^*$) a shorter input string. Thus, we can prove the correctness of Algorithm 6.10 on inputs of the form (e, x) by induction over $\min\{|e|, |x|\}$. The base case is when either $x =$ "" or e is a single alphabet symbol, "" or $\emptyset$. In the case the expression is of the form $e = (e'|e'')$ or $e = (e')(e'')$, we make recursive calls with the shorter expressions e', e''. In the case the expression is of the form $e = (e')^*$, we make recursive calls with either a shorter string

x and the same expression, or with the shorter expression e' and a string x' that is equal in length or shorter than x.

Solved Exercise 6.3 — Match the empty string. Give an algorithm that on input a regular expression e, outputs 1 if and only if $\Phi_e(\texttt{""}) = 1$.

■

Solution:

We can obtain such a recursive algorithm by using the following observations:

1. An expression of the form "" or $(e')^*$ always matches the empty string.
2. An expression of the form σ, where $\sigma \in \Sigma$ is an alphabet symbol, never matches the empty string.
3. The regular expression $\emptyset$ does not match the empty string.
4. An expression of the form $e'|e''$ matches the empty string if and only if one of e' or e'' matches it.
5. An expression of the form $(e')(e'')$ matches the empty string if and only if both e' and e'' match it.

Given the above observations, we see that the following algorithm will check if e matches the empty string:

Algorithm 6.11 — Check for empty string.

Input: Regular expression e over Σ^*, $x \in \Sigma^*$
Output: 1 iff e matches the emptry string.

```
1: procedure MatchEmpty(e)
2:     if e = "" then return 1 ;
3:     if e = ∅ or e ∈ Σ then return 0 ;
4:     if e = (e'|e'') then return MatchEmpty(e') or
   MatchEmpty(e'') ;
5:     if e = (e')(e'') then return MatchEmpty(e')
   and MatchEmpty(e'') ;
6:     if e = (e')* then return 1 ;
7: end procedure
```

■

6.4 EFFICIENT MATCHING OF REGULAR EXPRESSIONS (OPTIONAL)

Algorithm 6.10 is not very efficient. For example, given an expression involving concatenation or the "star" operation and a string of length

n, it can make n recursive calls, and hence it can be shown that in the worst case Algorithm 6.10 can take time *exponential* in the length of the input string x. Fortunately, it turns out that there is a much more efficient algorithm that can match regular expressions in *linear* (i.e., $O(n)$) time. Since we have not yet covered the topics of time and space complexity, we describe this algorithm in high level terms, without making the computational model precise. Rather we will use the colloquial notion of $O(n)$ running time as used in introduction to programming courses and whiteboard coding interviews. We will see a formal definition of time complexity in Chapter 13.

Theorem 6.12 — Matching regular expressions in linear time. Let e be a regular expression. Then there is an $O(n)$ time algorithm that computes Φ_e.

The implicit constant in the $O(n)$ term of Theorem 6.12 depends on the expression e. Thus, another way to state Theorem 6.12 is that for every expression e, there is some constant c and an algorithm A that computes Φ_e on n-bit inputs using at most $c \cdot n$ steps. This makes sense since in practice we often want to compute $\Phi_e(x)$ for a small regular expression e and a large document x. Theorem 6.12 tells us that we can do so with running time that scales linearly with the size of the document, even if it has (potentially) worse dependence on the size of the regular expression.

We prove Theorem 6.12 by obtaining more efficient recursive algorithm, that determines whether e matches a string $x \in \{0,1\}^n$ by reducing this task to determining whether a related expression e' matches $x_0, \ldots, x_{n-2}$. This will result in an expression for the running time of the form $T(n) = T(n-1) + O(1)$ which solves to $T(n) = O(n)$.

Restrictions of regular expressions. The central definition for the algorithm behind Theorem 6.12 is the notion of a *restriction* of a regular expression. The idea is that for every regular expression e and symbol σ in its alphabet, it is possible to define a regular expression $e[\sigma]$ such that $e[\sigma]$ matches a string x if and only if e matches the string $x\sigma$. For example, if e is the regular expression $(01)^*(01)$ (i.e., one or more occurrences of 01) then $e[1]$ is equal to $(01)^*0$ and $e[0]$ will be $\emptyset$. (Can you see why?)

Algorithm 6.13 computes the restriction $e[\sigma]$ given a regular expression e and an alphabet symbol σ. It always terminates, since the recursive calls it makes are always on expressions smaller than the input expression. Its correctness can be proven by induction on the length of the regular expression e, with the base cases being when e is "", $\emptyset$, or a single alphabet symbol τ.

Algorithm 6.13 — Restricting regular expression.

Input: Regular expression e over Σ, symbol $\sigma \in \Sigma$
Output: Regular expression $e' = e[\sigma]$ such that $\Phi_{e'}(x) = \Phi_e(x\sigma)$ for every $x \in \Sigma^*$

1: **procedure** RESTRICT(e,σ)
2: **if** $e =$ "" or $e = \emptyset$ **then return** $\emptyset$;
3: **if** $e = \tau$ for $\tau \in \Sigma$ **then return** "" if $\tau = \sigma$ and **return** $\emptyset$ otherwise ;
4: **if** $e = (e'|e'')$ **then return** $(\text{RESTRICT}(e',\sigma)|\text{RESTRICT}(e'',\sigma))$;
5: **if** $e = (e')^*$ **then return** $(e')^*(\text{RESTRICT}(e',\sigma))$;
6: **if** $e = (e')(e'')$ and $\Phi_{e''}(\text{""}) = 0$ **then return** $(e')(\text{RESTRICT}(e'',\sigma))$;
7: **if** $e = (e')(e'')$ and $\Phi_{e''}(\text{""}) = 1$ **then return** $(e')(\text{RESTRICT}(e'',\sigma) \mid \text{RESTRICT}(e',\sigma))$;
8: **end procedure**

Using this notion of restriction, we can define the following recursive algorithm for regular expression matching:

Algorithm 6.14 — Regular expression matching in linear time.

Input: Regular expression e over Σ^*, $x \in \Sigma^n$ where $n \in \mathbb{N}$
Output: $\Phi_e(x)$

1: **procedure** FMATCH(e,x)
2: **if** $x =$ "" **then return** MATCHEMPTY$(()e)$;
3: Let $e' \leftarrow \text{RESTRICT}(e, x_{n-1})$
4: **return** FMATCH$(e', x_0 \cdots x_{n-2})$
5: **end procedure**

By the definition of a restriction, for every $\sigma \in \Sigma$ and $x' \in \Sigma^*$, the expression e matches $x'\sigma$ if and only if $e[\sigma]$ matches x'. Hence for every e and $x \in \Sigma^n$, $\Phi_{e[x_{n-1}]}(x_0 \cdots x_{n-2}) = \Phi_e(x)$ and Algorithm 6.14 does return the correct answer. The only remaining task is to analyze its *running time*. Note that Algorithm 6.14 uses the MATCHEMPTY procedure of Solved Exercise 6.3 in the base case that $x =$ "". However, this is OK since this procedure's running time depends only on e and is independent of the length of the original input.

For simplicity, let us restrict our attention to the case that the alphabet Σ is equal to $\{0,1\}$. Define $C(\ell)$ to be the maximum number of operations that Algorithm 6.13 takes when given as input a regular expression e over $\{0,1\}$ of at most ℓ symbols. The value $C(\ell)$ can be shown to be polynomial in ℓ, though this is not important for this theorem, since we only care about the dependence of the time to compute

$\Phi_e(x)$ on the length of x and not about the dependence of this time on the length of e.

Algorithm 6.14 is a recursive algorithm that input an expression e and a string $x \in \{0,1\}^n$, does computation of at most $C(|e|)$ steps and then calls itself with input some expression e' and a string x' of length $n-1$. It will terminate after n steps when it reaches a string of length 0. So, the running time $T(e,n)$ that it takes for Algorithm 6.14 to compute Φ_e for inputs of length n satisfies the recursive equation:

$$T(e,n) = \max\{T(e[0],n-1), T(e[1],n-1)\} + C(|e|) \tag{6.2}$$

(In the base case $n=0$, $T(e,0)$ is equal to some constant depending only on e.) To get some intuition for the expression Eq. (6.2), let us open up the recursion for one level, writing $T(e,n)$ as

$$\begin{aligned} T(e,n) = \max\{&T(e[0][0],n-2) + C(|e[0]|),\\ &T(e[0][1],n-2) + C(|e[0]|),\\ &T(e[1][0],n-2) + C(|e[1]|),\\ &T(e[1][1],n-2) + C(|e[1]|)\} + C(|e|)\ . \end{aligned}$$

Continuing this way, we can see that $T(e,n) \leq n \cdot C(L) + O(1)$ where L is the largest length of any expression e' that we encounter along the way. Therefore, the following claim suffices to show that Algorithm 6.14 runs in $O(n)$ time:

Claim: Let e be a regular expression over $\{0,1\}$, then there is a number $L(e) \in \mathbb{N}$, such that for every sequence of symbols $\alpha_0, \ldots, \alpha_{n-1}$, if we define $e' = e[\alpha_0][\alpha_1]\cdots[\alpha_{n-1}]$ (i.e., restricting e to α_0, and then α_1 and so on and so forth), then $|e'| \leq L(e)$.

Proof of claim: For a regular expression e over $\{0,1\}$ and $\alpha \in \{0,1\}^m$, we denote by $e[\alpha]$ the expression $e[\alpha_0][\alpha_1]\cdots[\alpha_{m-1}]$ obtained by restricting e to α_0 and then to α_1 and so on. We let $S(e) = \{e[\alpha] | \alpha \in \{0,1\}^*\}$. We will prove the claim by showing that for every e, the set $S(e)$ is finite, and hence so is the number $L(e)$ which is the maximum length of e' for $e' \in S(e)$.

We prove this by induction on the structure of e. If e is a symbol, the empty string, or the empty set, then this is straightforward to show as the most expressions $S(e)$ can contain are the expression itself, "", and $\emptyset$. Otherwise we split to the two cases **(i)** $e = e'^*$ and **(ii)** $e = e'e''$, where e', e'' are smaller expressions (and hence by the induction hypothesis $S(e')$ and $S(e'')$ are finite). In the case **(i)**, if $e = (e')^*$ then $e[\alpha]$ is either equal to $(e')^* e'[\alpha]$ or it is simply the empty set if $e'[\alpha] = \emptyset$. Since $e'[\alpha]$ is in the set $S(e')$, the number of distinct expressions in $S(e)$ is at most $|S(e')| + 1$. In the case **(ii)**, if $e = e'e''$ then all the restrictions of e to strings α will either have the form $e'e''[\alpha]$ or the form $e'e''[\alpha]|e'[\alpha']$ where α' is some string such that $\alpha = \alpha'\alpha''$ and $e''[\alpha'']$

matches the empty string. Since $e''[\alpha] \in S(e'')$ and $e'[\alpha'] \in S(e')$, the number of the possible distinct expressions of the form $e[\alpha]$ is at most $|S(e'')| + |S(e'')| \cdot |S(e')|$. This completes the proof of the claim.

The bottom line is that while running Algorithm 6.14 on a regular expression e, all the expressions we ever encounter are in the finite set $S(e)$, no matter how large the input x is, and so the running time of Algorithm 6.14 satisfies the equation $T(n) = T(n-1) + C'$ for some constant C' depending on e. This solves to $O(n)$ where the implicit constant in the O notation can (and will) depend on e but crucially, not on the length of the input x.

6.4.1 Matching regular expressions using DFAs

Theorem 6.12 is already quite impressive, but we can do even better. Specifically, no matter how long the string x is, we can compute $\Phi_e(x)$ by maintaining only a constant amount of memory and moreover making a *single pass* over x. That is, the algorithm will scan the input x once from start to finish, and then determine whether or not x is matched by the expression e. This is important in the common case of trying to match a short regular expression over a huge file or document that might not even fit in our computer's memory. Of course, as we have seen before, a single-pass constant-memory algorithm is simply a deterministic finite automaton. As we will see in Theorem 6.17, a function can be computed by regular expression *if and only if* it can be computed by a DFA. We start with showing the "only if" direction:

Theorem 6.15 — DFA for regular expression matching. Let e be a regular expression. Then there is an algorithm that on input $x \in \{0,1\}^*$ computes $\Phi_e(x)$ while making a single pass over x and maintaining a constant amount of memory.

Proof Idea:

The single-pass constant-memory for checking if a string matches a regular expression is presented in Algorithm 6.16. The idea is to replace the recursive algorithm of Algorithm 6.14 with a dynamic program, using the technique of memoization. If you haven't taken yet an algorithms course, you might not know these techniques. This is OK; while this more efficient algorithm is crucial for the many practical applications of regular expressions, it is not of great importance for this book.

⋆

Algorithm 6.16 — Regular expression matching by a DFA.

Input: Regular expression e over Σ^*, $x \in \Sigma^n$ where $n \in \mathbb{N}$
Output: $\Phi_e(x)$

```
1: procedure DFAMatch(e,x)
2:     Let S ← S(e) be the set {e[α]|α ∈ {0,1}*} as defined
   in the proof of the linear-time matching theorem.
3:     for e' ∈ S do
4:         Let v_{e'} ← 1 if Φ_{e'}("") = 1 and v_{e'} ← 0 otherwise
5:     end for
6:     for i ∈ [n] do
7:         Let last_{e'} ← v_{e'} for all e' ∈ S
8:         Let v_{e'} ← last_{e'[x_i]} for all e' ∈ S
9:     end for
10:    return v_e
11: end procedure
```

Proof of Theorem 6.15. Algorithm 6.16 checks if a given string $x \in \Sigma^*$ is matched by the regular expression e. For every regular expression e, this algorithm has a constant number of Boolean variables (specifically a variable $v_{e'}$ for every $e' \in S(e)$ and a variable $last_{e'}$ for every e' in $S(e)$, using the fact that $e'[x_i]$ is in $S(e)$ for every $e' \in S(e)$). It makes a single pass over the input string. Hence it corresponds to a DFA. We prove its correctness by induction on the length n of the input. Specifically, we will argue that before reading x_i, the variable $v_{e'}$ is equal to $\Phi_{e'}(x_0 \cdots x_{i-1})$ for every $e' \in S(e)$. In the case $i = 0$ this holds since we initialize $v_{e'} = \Phi_{e'}(\texttt{""})$ for all $e' \in S(e)$. For $i > 0$ this holds by induction since the inductive hypothesis implies that $last'_e = \Phi_{e'}(x_0 \cdots x_{i-2})$ for all $e' \in S(e)$ and by the definition of the set $S(e')$, for every $e' \in S(e)$ and $x_{i-1} \in \Sigma$, $e'' = e'[x_{i-1}]$ is in $S(e)$ and $\Phi_{e'}(x_0 \cdots x_{i-1}) = \Phi_{e''}(x_0 \cdots x_i)$.

■

6.4.2 Equivalence of regular expressions and automata

Recall that a Boolean function $F : \{0,1\}^* \to \{0,1\}$ is defined to be *regular* if it is equal to Φ_e for some regular expression e. (Equivalently, a language $L \subseteq \{0,1\}^*$ is defined to be *regular* if there is a regular expression e such that e matches x iff $x \in L$.) The following theorem is the central result of automata theory:

Theorem 6.17 — DFA and regular expression equivalency. Let $F : \{0,1\}^* \to \{0,1\}$. Then F is regular if and only if there exists a DFA $(T, \mathcal{S})$ that computes F.

Proof Idea:

One direction follows from Theorem 6.15, which shows that for every regular expression e, the function Φ_e can be computed by a DFA (see for example Fig. 6.6). For the other direction, we show that given a DFA $(T, \mathcal{S})$ for every $v, w \in [C]$ we can find a regular expression that would match $x \in \{0,1\}^*$ if and only if the DFA starting in state v, will end up in state w after reading x.

⋆

Proof of Theorem 6.17. Since Theorem 6.15 proves the "only if" direction, we only need to show the "if" direction. Let $A = (T, \mathcal{S})$ be a DFA with C states that computes the function F. We need to show that F is regular.

For every $v, w \in [C]$, we let $F_{v,w} : \{0,1\}^* \to \{0,1\}$ be the function that maps $x \in \{0,1\}^*$ to 1 if and only if the DFA A, starting at the state v, will reach the state w if it reads the input x. We will prove that $F_{v,w}$ is regular for every v, w. This will prove the theorem, since by Definition 6.2, $F(x)$ is equal to the OR of $F_{0,w}(x)$ for every $w \in \mathcal{S}$. Hence if we have a regular expression for every function of the form $F_{v,w}$ then (using the | operation), we can obtain a regular expression for F as well.

To give regular expressions for the functions $F_{v,w}$, we start by defining the following functions $F^t_{v,w}$: for every $v, w \in [C]$ and $0 \leq t \leq C$, $F^t_{v,w}(x) = 1$ if and only if starting from v and observing x, the automata reaches w *with all intermediate states being in the set* $[t] = \{0, \ldots, t-1\}$ (see Fig. 6.7). That is, while v, w themselves might be outside $[t]$, $F^t_{v,w}(x) = 1$ if and only if throughout the execution of the automaton on the input x (when initiated at v) it never enters any of the states outside $[t]$ and still ends up at w. If $t = 0$ then $[t]$ is the empty set, and hence $F^0_{v,w}(x) = 1$ if and only if the automaton reaches w from v directly on x, without any intermediate state. If $t = C$ then all states are in $[t]$, and hence $F^t_{v,w} = F_{v,w}$.

We will prove the theorem by induction on t, showing that $F^t_{v,w}$ is regular for every v, w and t. For the **base case** of $t = 0$, $F^0_{v,w}$ is regular for every v, w since it can be described as one of the expressions "", $\emptyset$, 0, 1 or 0|1. Specifically, if $v = w$ then $F^0_{v,w}(x) = 1$ if and only if x is the empty string. If $v \neq w$ then $F^0_{v,w}(x) = 1$ if and only if x consists of a single symbol $\sigma \in \{0,1\}$ and $T(v, \sigma) = w$. Therefore in this case $F^0_{v,w}$ corresponds to one of the four regular expressions 0|1, 0, 1 or $\emptyset$, depending on whether A transitions to w from v when it reads either 0 or 1, only one of these symbols, or neither.

Inductive step: Now that we've seen the base case, let us prove the general case by induction. Assume, via the induction hypothesis, that for every $v', w' \in [C]$, we have a regular expression $R^t_{v',w'}$ that computes $F^t_{v',w'}$. We need to prove that $F^{t+1}_{v,w}$ is regular for every v, w.

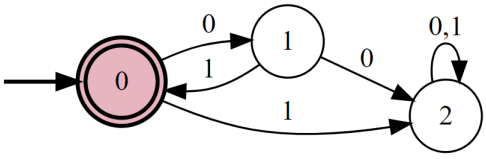


Figure 6.6: A deterministic finite automaton that computes the function $\Phi_{(01)^*}$.

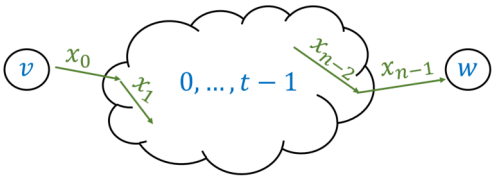


Figure 6.7: Given a DFA of C states, for every $v, w \in [C]$ and number $t \in \{0, \ldots, C\}$ we define the function $F^t_{v,w} : \{0,1\}^* \to \{0,1\}$ to output one on input $x \in \{0,1\}^*$ if and only if when the DFA is initialized in the state v and is given the input x, it will reach the state w while going only through the intermediate states $\{0, \ldots, t-1\}$.

If the automaton arrives from v to w using the intermediate states $[t+1]$, then it visits the t-th state zero or more times. If the path labeled by x causes the automaton to get from v to w without visiting the t-th state at all, then x is matched by the regular expression $R^t_{v,w}$. If the path labeled by x causes the automaton to get from v to w while visiting the t-th state $k > 0$ times, then we can think of this path as:

- First travel from v to t using only intermediate states in $[t-1]$.
- Then go from t back to itself $k-1$ times using only intermediate states in $[t-1]$
- Then go from t to w using only intermediate states in $[t-1]$.

Therefore in this case the string x is matched by the regular expression $R^t_{v,t}(R^t_{t,t})^* R^t_{t,w}$. (See also Fig. 6.8.)

Therefore we can compute $F^{t+1}_{v,w}$ using the regular expression

$$R^t_{v,w} \mid R^t_{v,t}(R^t_{t,t})^* R^t_{t,w} \ .$$

This completes the proof of the inductive step and hence of the theorem.

■

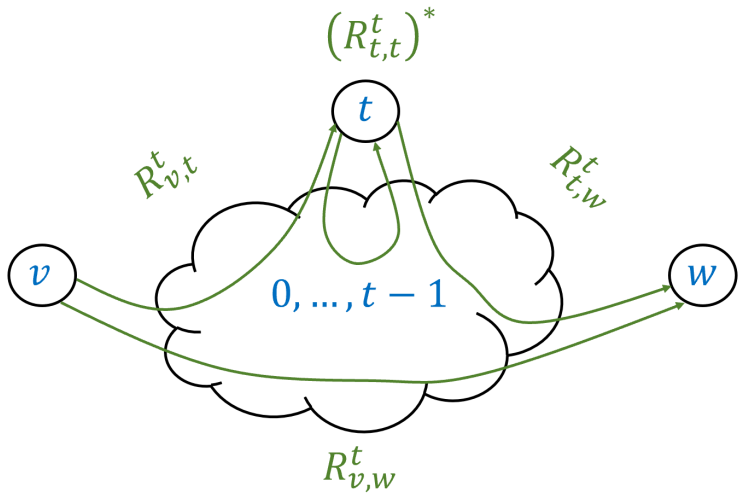


Figure 6.8: If we have regular expressions $R^t_{v',w'}$ corresponding to $F^t_{v',w'}$ for every $v', w' \in [C]$, we can obtain a regular expression $R^{t+1}_{v,w}$ corresponding to $F^{t+1}_{v,w}$. The key observation is that a path from v to w using $\{0, \ldots, t\}$ either does not touch t at all, in which case it is captured by the expression $R^t_{v,w}$, or it goes from v to t, comes back to t zero or more times, and then goes from t to w, in which case it is captured by the expression $R^t_{v,t}(R^t_{t,t})^* R^t_{t,w}$.

6.4.3 Closure properties of regular expressions

If F and G are regular functions computed by the expressions e and f respectively, then the expression $e|f$ computes the function $H = F \vee G$ defined as $H(x) = F(x) \vee G(x)$. Another way to say this is that the set of regular functions is *closed under the OR operation*. That is, if F and G are regular then so is $F \vee G$. An important corollary of Theorem 6.17 is that this set is also closed under the NOT operation:

Lemma 6.18 — Regular expressions closed under complement. If $F : \{0,1\}^* \to \{0,1\}$ is regular then so is the function $\overline{F}$, where $\overline{F}(x) = 1 - F(x)$ for every $x \in \{0,1\}^*$.

Proof. If F is regular then by Theorem 6.12 it can be computed by a DFA A. But we can then construct a DFA $\overline{A}$ which does the same computation but flips the set of accepted states. The DFA $\overline{A}$ will compute $\overline{F}$. By Theorem 6.17 this implies that $\overline{F}$ is regular as well. ■

Since $a \wedge b = \overline{\overline{a} \vee \overline{b}}$, Lemma 6.18 implies that the set of regular functions is closed under the AND operation as well. Moreover, since OR, NOT and AND are a universal basis, this set is also closed under NAND, XOR, and any other finite function. That is, we have the following corollary:

Theorem 6.19 — Closure of regular expressions. Let $f : \{0,1\}^k \to \{0,1\}$ be any finite Boolean function, and let $F_0, \ldots, F_{k-1} : \{0,1\}^* \to \{0,1\}$ be regular functions. Then the function $G(x) = f(F_0(x), F_1(x), \ldots, F_{k-1}(x))$ is regular.

Proof. This is a direct consequence of the closure of regular functions under OR and NOT (and hence AND), combined with Theorem 4.13, that states that every f can be computed by a Boolean circuit (which is simply a combination of the AND, OR, and NOT operations). ■

6.5 LIMITATIONS OF REGULAR EXPRESSIONS AND THE PUMPING LEMMA

The efficiency of regular expression matching makes them very useful. This is why operating systems and text editors often restrict their search interface to regular expressions and do not allow searching by specifying an arbitrary function. However, this efficiency comes at a cost. As we have seen, regular expressions cannot compute every function. In fact, there are some very simple (and useful!) functions that they cannot compute. Here is one example:

Lemma 6.20 — Matching parentheses. Let $\Sigma = \{(,)\}$ and $MATCHPAREN : \Sigma^* \to \{0,1\}$ be the function that given a string of parentheses, outputs 1 if and only if every opening parenthesis is matched by a corresponding closed one. Then there is no regular expression over Σ that computes $MATCHPAREN$.

Lemma 6.20 is a consequence of the following result, which is known as the *pumping lemma*:

Theorem 6.21 — Pumping Lemma. Let e be a regular expression over some alphabet Σ. Then there is some number n_0 such that for every $w \in \Sigma^*$ with $|w| > n_0$ and $\Phi_e(w) = 1$, we can write $w = xyz$ for strings $x, y, z \in \Sigma^*$ satisfying the following conditions:

1. $|y| \geq 1$.
2. $|xy| \leq n_0$.
3. $\Phi_e(xy^k z) = 1$ for every $k \in \mathbb{N}$.

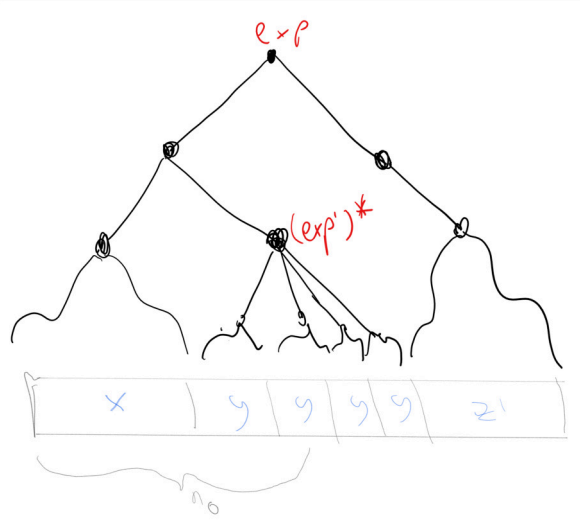


Figure 6.9: To prove the "pumping lemma" we look at a word w that is much larger than the regular expression e that matches it. In such a case, part of w must be matched by some sub-expression of the form $(e')^*$, since this is the only operator that allows matching words longer than the expression. If we look at the "leftmost" such sub-expression and define y^k to be the string that is matched by it, we obtain the partition needed for the pumping lemma.

Proof Idea:

The idea behind the proof is the following. Let n_0 be twice the number of symbols that are used in the expression e, then the only way that there is some w with $|w| > n_0$ and $\Phi_e(w) = 1$ is that e contains the $*$ (i.e. star) operator and that there is a non-empty substring y of w that was matched by $(e')^*$ for some sub-expression e' of e. We can now repeat y any number of times and still get a matching string. See also Fig. 6.9.

⋆

The pumping lemma is a bit cumbersome to state, but one way to remember it is that it simply says the following: *"if a string matching a regular expression is long enough, one of its substrings must be matched using the $*$ operator"*.

Proof of Theorem 6.21. To prove the lemma formally, we use induction on the length of the expression. Like all induction proofs, this will be somewhat lengthy, but at the end of the day it directly follows the intuition above that *somewhere* we must have used the star operation. Reading this proof, and in particular understanding how the formal proof below corresponds to the intuitive idea above, is a very good way to get more comfortable with inductive proofs of this form.

Our inductive hypothesis is that for an n length expression, $n_0 = 2n$ satisfies the conditions of the lemma. The **base case** is when the expression is a single symbol $\sigma \in \Sigma$ or that the expression is $\emptyset$ or "". In all these cases the conditions of the lemma are satisfied simply because $n_0 = 2$, and there exists no string x of length larger than n_0 that is matched by the expression.

We now prove the **inductive step**. Let e be a regular expression with $n > 1$ symbols. We set $n_0 = 2n$ and let $w \in \Sigma^*$ be a string satisfying $|w| > n_0$. Since e has more than one symbol, it has one of the forms **(a)** $e'|e''$, **(b)**, $(e')(e'')$, or **(c)** $(e')^*$ where in all these cases the subexpressions e' and e'' have fewer symbols than e and hence satisfy the induction hypothesis.

In the case **(a)**, every string w matched by e must be matched by either e' or e''. If e' matches w then, since $|w| > 2|e'|$, by the induction hypothesis there exist x, y, z with $|y| \geq 1$ and $|xy| \leq 2|e'| < n_0$ such that e' (and therefore also $e = e'|e''$) matches $xy^k z$ for every k. The same arguments works in the case that e'' matches w.

In the case **(b)**, if w is matched by $(e')(e'')$ then we can write $w = w'w''$ where e' matches w' and e'' matches w''. We split to subcases. If $|w'| > 2|e'|$ then by the induction hypothesis there exist x, y, z' with $|y| \geq 1$, $|xy| \leq 2|e'| < n_0$ such that $w' = xyz'$ and e' matches $xy^k z'$ for every $k \in \mathbb{N}$. This completes the proof since if we set $z = z'w''$ then we see that $w = w'w'' = xyz$ and $e = (e')(e'')$ matches $xy^k z$ for every $k \in \mathbb{N}$. Otherwise, if $|w'| \leq 2|e'|$ then since $|w| = |w'| + |w''| > n_0 = 2(|e'| + |e''|)$, it must be that $|w''| > 2|e''|$. Hence by the induction hypothesis there exist x', y, z such that $|y| \geq 1$, $|x'y| \leq 2|e''|$ and e'' matches $x'y^k z$ for every $k \in \mathbb{N}$. But now if we set $x = w'x'$ we see that $|xy| = |w'| + |x'y| \leq 2|e'| + 2|e''| = n_0$ and on the other hand the expression $e = (e')(e'')$ matches $xy^k z = w'x'y^k z$ for every $k \in \mathbb{N}$.

In case **(c)**, if w is matched by $(e')^*$ then $w = w_0 \cdots w_t$ where for every $i \in [t]$, w_i is a nonempty string matched by e'. If $|w_0| > 2|e'|$, then we can use the same approach as in the concatenation case above. Otherwise, we simply note that if x is the empty string, $y = w_0$, and $z = w_1 \cdots w_t$ then $|xy| \leq n_0$ and $xy^k z$ is matched by $(e')^*$ for every $k \in \mathbb{N}$.

■

R

Remark 6.22 — Recursive definitions and inductive proofs. When an object is *recursively defined* (as in the case of regular expressions) then it is natural to prove properties of such objects by *induction*. That is, if we want to prove that all objects of this type have property P, then it is natural to use an inductive step that says that if o', o'', o''' etc have property P then so is an object o that is obtained by composing them.

Using the pumping lemma, we can easily prove Lemma 6.20 (i.e., the non-regularity of the "matching parenthesis" function):

Proof of Lemma 6.20. Suppose, towards the sake of contradiction, that there is an expression e such that $\Phi_e = MATCHPAREN$. Let n_0 be the number obtained from Theorem 6.21 and let $w = \langle^{n_0}\rangle^{n_0}$ (i.e., n_0 left parenthesis followed by n_0 right parenthesis). Then we see that if we write $w = xyz$ as in Theorem 6.21, the condition $|xy| \leq n_0$ implies that y consists solely of left parenthesis. Hence the string xy^2z will contain more left parenthesis than right parenthesis. Hence $MATCHPAREN(xy^2z) = 0$ but by the pumping lemma $\Phi_e(xy^2z) = 1$, contradicting our assumption that $\Phi_e = MATCHPAREN$. ■

The pumping lemma is a very useful tool to show that certain functions are *not* computable by a regular expression. However, it is *not* an "if and only if" condition for regularity: there are non-regular functions that still satisfy the pumping lemma conditions. To understand the pumping lemma, it is crucial to follow the order of quantifiers in Theorem 6.21. In particular, the number n_0 in the statement of Theorem 6.21 depends on the regular expression (in the proof we chose n_0 to be twice the number of symbols in the expression). So, if we want to use the pumping lemma to rule out the existence of a regular expression e computing some function F, we need to be able to choose an appropriate input $w \in \{0,1\}^*$ that can be arbitrarily large and satisfies $F(w) = 1$. This makes sense if you think about the intuition behind the pumping lemma: we need w to be large enough as to force the use of the star operator.

Solved Exercise 6.4 — Palindromes is not regular. Prove that the following function over the alphabet $\{0,1,;\}$ is not regular: $PAL(w) = 1$ if and only if $w = u;u^R$ where $u \in \{0,1\}^*$ and u^R denotes u "reversed": the string $u_{|u|-1}\cdots u_0$. (The *Palindrome* function is most often defined without an explicit separator character ;, but the version with such a separator is a bit cleaner, and so we use it here. This does not make

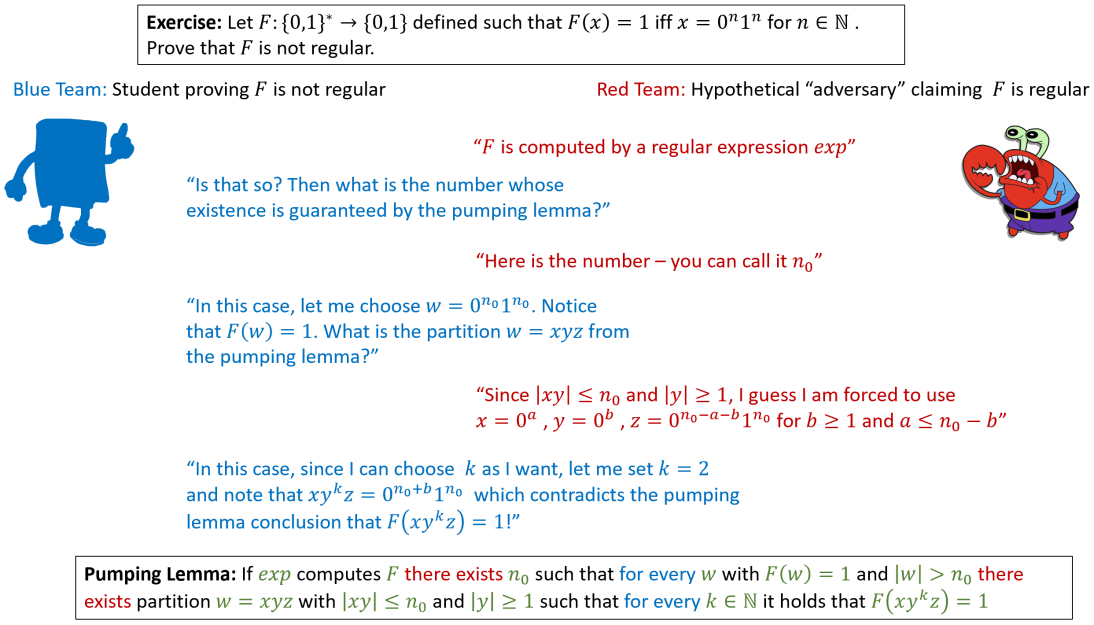


Figure 6.10: A cartoon of a proof using the pumping lemma that a function F is not regular. The pumping lemma states that if F is regular then *there exists* a number n_0 such that *for every* large enough w with $F(w) = 1$, *there exists* a partition of w to $w = xyz$ satisfying certain conditions such that *for every* $k \in \mathbb{N}$, $F(xy^kz) = 1$. You can imagine a pumping-lemma based proof as a game between you and the adversary. Every *there exists* quantifier corresponds to an object you are free to choose on your own (and base your choice on previously chosen objects). Every *for every* quantifier corresponds to an object the adversary can choose arbitrarily (and again based on prior choices) as long as it satisfies the conditions. A valid proof corresponds to a strategy by which no matter what the adversary does, you can win the game by obtaining a contradiction which would be a choice of k that would result in $F(xy^kz) = 0$, hence violating the conclusion of the pumping lemma.

much difference, as one can easily encode the separator as a special binary string instead.)

Solution:

We use the pumping lemma. Suppose toward the sake of contradiction that there is a regular expression e computing PAL, and let n_0 be the number obtained by the pumping lemma (Theorem 6.21). Consider the string $w = 0^{n_0};0^{n_0}$. Since the reverse of the all zero string is the all zero string, $PAL(w) = 1$. Now, by the pumping lemma, if PAL is computed by e, then we can write $w = xyz$ such that $|xy| \leq n_0$, $|y| \geq 1$ and $PAL(xy^kz) = 1$ for every $k \in \mathbb{N}$. In particular, it must hold that $PAL(xz) = 1$, but this is a contradiction, since $xz = 0^{n_0-|y|};0^{n_0}$ and so its two parts are not of the same length and in particular are not the reverse of one another.

For yet another example of a pumping-lemma based proof, see Fig. 6.10 which illustrates a cartoon of the proof of the non-regularity of the function $F : \{0,1\}^* \to \{0,1\}$ which is defined as $F(x) = 1$ iff $x = 0^n1^n$ for some $n \in \mathbb{N}$ (i.e., x consists of a string of consecutive zeroes, followed by a string of consecutive ones of the same length).

6.6 ANSWERING SEMANTIC QUESTIONS ABOUT REGULAR EXPRESSIONS

Regular expressions have applications beyond search. For example, regular expressions are often used to define *tokens* (such as what is a valid variable identifier, or keyword) in the design of *parsers*, *compilers* and *interpreters* for programming languages. Regular expressions have other applications too: for example, in recent years, the world of networking moved from fixed topologies to "software defined networks". Such networks are routed by programmable switches that can implement *policies* such as "if packet is secured by SSL then forward it to A, otherwise forward it to B". To represent such policies we need a language that is on one hand sufficiently expressive to capture the policies we want to implement, but on the other hand sufficiently restrictive so that we can quickly execute them at network speed and also be able to answer questions such as "can C see the packets moved from A to B?". The NetKAT network programming language uses a variant of regular expressions to achieve precisely that. For this application, it is important that we are not merely able to answer whether an expression e matches a string x but also answer *semantic questions* about regular expressions such as "do expressions

e and e' compute the same function?" and "does there exist a string x that is matched by the expression e?". The following theorem shows that we can answer the latter question:

Theorem 6.23 — Emptiness of regular languages is computable. There is an algorithm that given a regular expression e, outputs 1 if and only if Φ_e is the constant zero function.

Proof Idea:

The idea is that we can directly observe this from the structure of the expression. The only way a regular expression e computes the constant zero function is if e has the form $\emptyset$ or is obtained by concatenating $\emptyset$ with other expressions.

⋆

Proof of Theorem 6.23. Define a regular expression to be "empty" if it computes the constant zero function. Given a regular expression e, we can determine if e is empty using the following rules:

- If e has the form σ or "" then it is not empty.
- If e is not empty then $e|e'$ is not empty for every e'.
- If e is not empty then e^* is not empty.
- If e and e' are both not empty then $e\ e'$ is not empty.
- $\emptyset$ is empty.

Using these rules, it is straightforward to come up with a recursive algorithm to determine emptiness.

■

Using Theorem 6.23, we can obtain an algorithm that determines whether or not two regular expressions e and e' are *equivalent*, in the sense that they compute the same function.

Theorem 6.24 — Equivalence of regular expressions is computable. Let $REGEQ : \{0,1\}^* \to \{0,1\}$ be the function that on input (a string representing) a pair of regular expressions e, e', $REGEQ(e,e') = 1$ if and only if $\Phi_e = \Phi_{e'}$. Then there is an algorithm that computes $REGEQ$.

Proof Idea:

The idea is to show that given a pair of regular expressions e and e' we can find an expression e'' such that $\Phi_{e''}(x) = 1$ if and only if $\Phi_e(x) \neq \Phi_{e'}(x)$. Therefore $\Phi_{e''}$ is the constant zero function if and only

if e and e' are equivalent, and thus we can test for emptiness of e'' to determine equivalence of e and e'.

⋆

Proof of Theorem 6.24. We will prove Theorem 6.24 from Theorem 6.23. (The two theorems are in fact equivalent: it is easy to prove Theorem 6.23 from Theorem 6.24, since checking for emptiness is the same as checking equivalence with the expression $\emptyset$.) Given two regular expressions e and e', we will compute an expression e'' such that $\Phi_{e''}(x) = 1$ if and only if $\Phi_e(x) \neq \Phi_{e'}(x)$. One can see that e is equivalent to e' if and only if e'' is empty.

We start with the observation that for every bit $a, b \in \{0,1\}$, $a \neq b$ if and only if

$$(a \wedge \overline{b}) \ \vee \ (\overline{a} \wedge b) \ .$$

Hence we need to construct e'' such that for every x,

$$\Phi_{e''}(x) = (\Phi_e(x) \wedge \overline{\Phi_{e'}(x)}) \ \vee \ (\overline{\Phi_e(x)} \wedge \Phi_{e'}(x)) \ . \tag{6.3}$$

To construct the expression e'', we will show how given any pair of expressions e and e', we can construct expressions $e \wedge e'$ and $\overline{e}$ that compute the functions $\Phi_e \wedge \Phi_{e'}$ and $\overline{\Phi_e}$ respectively. (Computing the expression for $e \vee e'$ is straightforward using the $|$ operation of regular expressions.)

Specifically, by Lemma 6.18, regular functions are closed under negation, which means that for every regular expression e, there is an expression $\overline{e}$ such that $\Phi_{\overline{e}}(x) = 1 - \Phi_e(x)$ for every $x \in \{0,1\}^*$. Now, for every two expressions e and e', the expression

$$e \wedge e' = \overline{(\overline{e}|\overline{e'})}$$

computes the AND of the two expressions. Given these two transformations, we see that for every regular expressions e and e' we can find a regular expression e'' satisfying (6.3) such that e'' is empty if and only if e and e' are equivalent.

■

Chapter Recap

- We model computational tasks on arbitrarily large inputs using *infinite* functions $F : \{0,1\}^* \to \{0,1\}^*$.
- Such functions take an arbitrarily long (but still finite!) string as input, and cannot be described by a finite table of inputs and outputs.
- A function with a single bit of output is known as a *Boolean function*, and the task of computing it is equivalent to deciding a *language* $L \subseteq \{0,1\}^*$.

- *Deterministic finite automata* (DFAs) are one simple model for computing (infinite) Boolean functions.
- There are some functions that cannot be computed by DFAs.
- The set of functions computable by DFAs is the same as the set of languages that can be recognized by regular expressions.

6.7 EXERCISES

Exercise 6.1 — Closure properties of regular functions. Suppose that $F, G : \{0,1\}^* \to \{0,1\}$ are regular. For each one of the following definitions of the function H, either prove that H is always regular or give a counterexample for regular F, G that would make H not regular.

1. $H(x) = F(x) \vee G(x)$.
2. $H(x) = F(x) \wedge G(x)$
3. $H(x) = NAND(F(x), G(x))$.
4. $H(x) = F(x^R)$ where x^R is the reverse of x: $x^R = x_{n-1}x_{n-2}\cdots x_o$ for $n = |x|$.
5. $H(x) = \begin{cases} 1 & x = uv \text{ s.t. } F(u) = G(v) = 1 \\ 0 & \text{otherwise} \end{cases}$
6. $H(x) = \begin{cases} 1 & x = uu \text{ s.t. } F(u) = G(u) = 1 \\ 0 & \text{otherwise} \end{cases}$
7. $H(x) = \begin{cases} 1 & x = uu^R \text{ s.t. } F(u) = G(u) = 1 \\ 0 & \text{otherwise} \end{cases}$

■

Exercise 6.2 One among the following two functions that map $\{0,1\}^*$ to $\{0,1\}$ can be computed by a regular expression, and the other one cannot. For the one that can be computed by a regular expression, write the expression that does it. For the one that cannot, prove that this cannot be done using the pumping lemma.

- $F(x) = 1$ if 4 divides $\sum_{i=0}^{|x|-1} x_i$ and $F(x) = 0$ otherwise.
- $G(x) = 1$ if and only if $\sum_{i=0}^{|x|-1} x_i \geq |x|/4$ and $G(x) = 0$ otherwise.

■

Exercise 6.3 — Non-regularity. 1. Prove that the following function $F : \{0,1\}^* \to \{0,1\}$ is not regular. For every $x \in \{0,1\}^*$, $F(x) = 1$ iff x is of the form $x = 1^{3^i}$ for some $i > 0$.

2. Prove that the following function $F : \{0,1\}^* \to \{0,1\}$ is not regular. For every $x \in \{0,1\}^*$, $F(x) = 1$ iff $\sum_j x_j = 3^i$ for some $i > 0$.

■

6.8 BIBLIOGRAPHICAL NOTES

The relation of regular expressions with finite automata is a beautiful topic, on which we only touch upon in this text. It is covered more extensively in [Sip97; HMU14; Koz97]. These texts also discuss topics such as *non-deterministic finite automata* (NFA) and the relation between context-free grammars and pushdown automata.

The automaton of Fig. 6.4 was generated using the FSM simulator of Ivan Zuzak and Vedrana Jankovic. Our proof of Theorem 6.12 is closely related to the Myhill-Nerode Theorem. One direction of the Myhill-Nerode theorem can be stated as saying that if e is a regular expression then there is at most a finite number of strings $z_0, \ldots, z_{k-1}$ such that $\Phi_{e[z_i]} \neq \Phi_{e[z_j]}$ for every $0 \leq i \neq j < k$.

Learning Objectives:

- Learn the model of *Turing machines*, which can compute functions of *arbitrary input lengths*.
- See a programming-language description of Turing machines, using NAND-TM programs, which add *loops* and *arrays* to NAND-CIRC.
- See some basic syntactic sugar and equivalence of variants of Turing machines and NAND-TM programs.

7
Loops and infinity

_ "The bounds of arithmetic were however outstepped the moment the idea of applying the [punched] cards had occurred; and the Analytical Engine does not occupy common ground with mere 'calculating machines.'... In enabling mechanism to combine together general symbols, in successions of unlimited variety and extent, a uniting link is established between the operations of matter and the abstract mental processes of the most abstract branch of mathematical science."_, Ada Augusta, countess of Lovelace, 1843

As the quote of Chapter 6 says, an algorithm is "a finite answer to an infinite number of questions". To express an algorithm, we need to write down a finite set of instructions that will enable us to compute on arbitrarily long inputs. To describe and execute an algorithm we need the following components (see Fig. 7.1):

- The finite set of instructions to be performed.
- Some "local variables" or finite state used in the execution.
- A potentially unbounded working memory to store the input and any other values we may require later.
- While the memory is unbounded, at every single step we can only read and write to a finite part of it, and we need a way to *address* which are the parts we want to read from and write to.
- If we only have a finite set of instructions but our input can be arbitrarily long, we will need to *repeat* instructions (i.e., *loop* back). We need a mechanism to decide when we will loop and when we will halt.

This chapter: A non-mathy overview

In this chapter, we give a general model of an algorithm, which (unlike Boolean circuits) is not restricted to a fixed

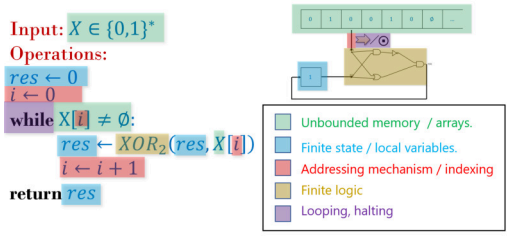


Figure 7.1: An algorithm is a finite recipe to compute on arbitrarily long inputs. The components of an algorithm include the instructions to be performed, finite state or "local variables", the memory to store the input and intermediate computations, as well as mechanisms to decide which part of the memory to access, and when to repeat instructions and when to halt.

input length, and (unlike finite automata) is not restricted to a finite amount of working memory. We will see two ways to model algorithms:

- *Turing machines*, invented by Alan Turing in 1936, are hypothetical abstract devices that yield finite descriptions of algorithms that can handle arbitrarily long inputs.
- The *NAND-TM Programming language* extends NAND-CIRC with the notion of *loops* and *arrays* to obtain finite programs that can compute a function with arbitrarily long inputs.

It turns out that these two models are *equivalent*. In fact, they are equivalent to many other computational models, including programming languages such as C, Lisp, Python, JavaScript, etc. This notion, known as *Turing equivalence* or *Turing completeness*, will be discussed in Chapter 8. See Fig. 7.2 for an overview of the models presented in this chapter and Chapter 8.

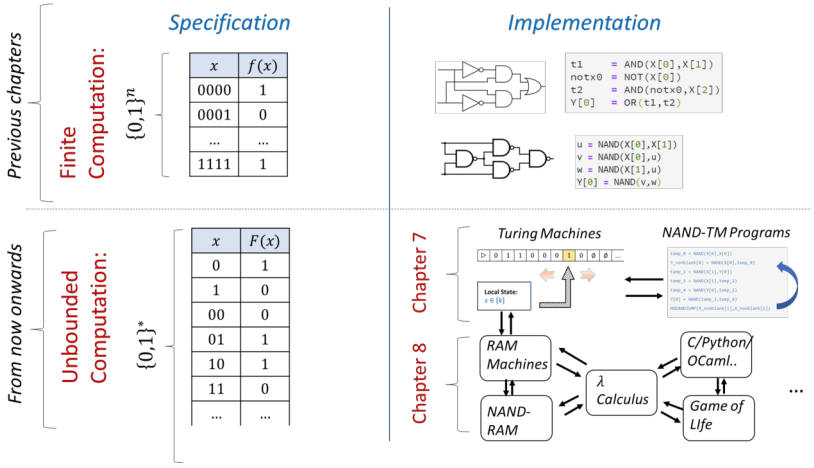


Figure 7.2: Overview of our models for finite and unbounded computation. In the previous chapters we study the computation of *finite functions*, which are functions $f : \{0,1\}^n \to \{0,1\}^m$ for some fixed n, m, and modeled computing these functions using circuits or straight-line programs. In this chapter we study computing *unbounded* functions of the form $F : \{0,1\}^* \to \{0,1\}^m$ or $F : \{0,1\}^* \to \{0,1\}^*$. We model computing these functions using *Turing Machines* or (equivalently) NAND-TM programs, which add the notion of *loops* to the NAND-CIRC programming language. In Chapter 8 we will show that these models are equivalent to many other models, including RAM machines, the λ calculus, and all the common programming languages including C, Python, Java, JavaScript, etc.

7.1 TURING MACHINES

" Computing is normally done by writing certain symbols on paper. We may suppose that this paper is divided into squares like a child's arithmetic book.. The behavior of the [human] computer at any moment is determined by the symbols which he is observing, and of his' state of mind' at that moment... We may suppose that in a simple operation not more than one symbol is altered.", "We compare a man in the process of computing ... to a machine which is only capable of a finite number of configurations... The machine is supplied with a

'tape' (*the analogue of paper*) *... divided into sections* (*called 'squares'*) *each capable of bearing a 'symbol' "*, Alan Turing, 1936

_ "What is the difference between a Turing machine and the modern computer? It's the same as that between Hillary's ascent of Everest and the establishment of a Hilton hotel on its peak."_ , Alan Perlis, 1982.

The "granddaddy" of all models of computation is the *Turing machine*. Turing machines were defined in 1936 by Alan Turing in an attempt to formally capture all the functions that can be computed by human "computers" (see Fig. 7.4) that follow a well-defined set of rules, such as the standard algorithms for addition or multiplication.

Turing thought of such a person as having access to as much "scratch paper" as they need. For simplicity, we can think of this scratch paper as a one dimensional piece of graph paper (or *tape*, as it is commonly referred to). The paper is divided into "cells", where each "cell" can hold a single symbol (e.g., one digit or letter, and more generally, some element of a finite *alphabet*). At any point in time, the person can read from and write to a single cell of the paper. Based on the contents of this cell, the person can update their finite mental state, and/or move to the cell immediately to the left or right of the current one.

Turing modeled such a computation by a "machine" that maintains one of k states. At each point in time the machine reads from its "work tape" a single symbol from a finite alphabet Σ and uses that to update its state, write to tape, and possibly move to an adjacent cell (see Fig. 7.7). To compute a function F using this machine, we initialize the tape with the input $x \in \{0,1\}^*$ and our goal is to ensure that the tape will contain the value $F(x)$ at the end of the computation. Specifically, a computation of a Turing machine M with k states and alphabet Σ on input $x \in \{0,1\}^*$ proceeds as follows:

- Initially the machine is at state 0 (known as the "starting state") and the tape is initialized to $\triangleright, x_0, \ldots, x_{n-1}, \varnothing, \varnothing, \ldots$. We use the symbol $\triangleright$ to denote the beginning of the tape, and the symbol $\varnothing$ to denote an empty cell. We will always assume that the alphabet Σ is a (potentially strict) superset of $\{\triangleright, \varnothing, 0, 1\}$.
- The location i to which the machine points to is set to 0.
- At each step, the machine reads the symbol $\sigma = T[i]$ that is in the ith location of the tape. Based on this symbol and its state s, the machine decides on:
  - What symbol σ' to write on the tape
  - Whether to move **L**eft (i.e., $i \leftarrow i-1$), **R**ight (i.e., $i \leftarrow i+1$), **S**tay in place, or **H**alt the computation.

Figure 7.3: Aside from his many other achievements, Alan Turing was an excellent long-distance runner who just fell shy of making England's Olympic team. A fellow runner once asked him why he punished himself so much in training. Alan said "I have such a stressful job that the only way I can get it out of my mind is by running hard; it's the only way I can get some release."

Figure 7.4: Until the advent of electronic computers, the word "computer" was used to describe a person that performed calculations. Most of these "human computers" were women, and they were absolutely essential to many achievements, including mapping the stars, breaking the Enigma cipher, and the NASA space mission; see also the bibliographical notes. Photo from National Photo Company Collection; see also [Sob17].

Figure 7.5: Steam-powered Turing machine mural, painted by CSE grad students at the University of Washington on the night before spring qualifying examinations, 1987. Image from https://www.cs.washington.edu/building/art/SPTM.

  - What is going to be the new state $s \in [k]$

- The set of rules the Turing machine follows is known as its *transition function*.

- When the machine halts, its output is the binary string obtained by reading the tape from the beginning until the first location in which it contains a $\varnothing$ symbol, and then outputting all 0 and 1 symbols in sequence, dropping the initial $\triangleright$ symbol if it exists, as well as the final $\varnothing$ symbol.

7.1.1 Extended example: A Turing machine for palindromes

Let PAL (for *palindromes*) be the function that on input $x \in \{0,1\}^*$, outputs 1 if and only if x is an (even length) *palindrome*, in the sense that $x = w_0 \cdots w_{n-1} w_{n-1} w_{n-2} \cdots w_0$ for some $n \in \mathbb{N}$ and $w \in \{0,1\}^n$.

We now show a Turing machine M that computes PAL. To specify M we need to specify **(i)** M's tape alphabet Σ which should contain at least the symbols 0,1, $\triangleright$ and $\varnothing$, and **(ii)** M's *transition function* which determines what action M takes when it reads a given symbol while it is in a particular state.

In our case, M will use the alphabet $\{0,1,\triangleright,\varnothing,\times\}$ and will have $k = 13$ states. Though the states are simply numbers between 0 and $k-1$, we will give them the following labels for convenience:

State	Label
0	START
1	RIGHT_0
2	RIGHT_1
3	LOOK_FOR_0
4	LOOK_FOR_1
5	RETURN
6	REJECT
7	ACCEPT
8	OUTPUT_0
9	OUTPUT_1
10	0_AND_BLANK
11	1_AND_BLANK
12	BLANK_AND_STOP

We describe the operation of our Turing machine M in words:

- M starts in state START and goes right, looking for the first symbol that is 0 or 1. If it finds $\varnothing$ before it hits such a symbol then it moves to the OUTPUT_1 state described below.

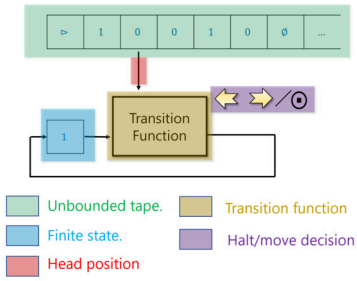


Figure 7.6: The components of a Turing Machine. Note how they correspond to the general components of algorithms as described in Fig. 7.1.

- Once M finds such a symbol $b \in \{0,1\}$, M deletes b from the tape by writing the $\times$ symbol, it enters either the `RIGHT_0` or `RIGHT_1` mode according to the value of b and starts moving rightwards until it hits the first $\varnothing$ or $\times$ symbol.
- Once M finds this symbol, it goes into the state `LOOK_FOR_0` or `LOOK_FOR_1` depending on whether it was in the state `RIGHT_0` or `RIGHT_1` and makes one left move.
- In the state `LOOK_FOR_`b, M checks whether the value on the tape is b. If it is, then M deletes it by changing its value to $\times$, and moves to the state `RETURN`. Otherwise, it changes to the `OUTPUT_0` state.
- The `RETURN` state means that M goes back to the beginning. Specifically, M moves leftward until it hits the first symbol that is not 0 or 1, in which case it changes its state to `START`.
- The `OUTPUT_`b states mean that M will eventually output the value b. In both the `OUTPUT_0` and `OUTPUT_1` states, M goes left until it hits $\triangleright$. Once it does so, it makes a right step, and changes to the `1_AND_BLANK` or `0_AND_BLANK` states respectively. In the latter states, M writes the corresponding value, moves right and changes to the `BLANK_AND_STOP` state, in which it writes $\varnothing$ to the tape and halts.

The above description can be turned into a table describing for each one of the $13 \cdot 5$ combination of state and symbol, what the Turing machine will do when it is in that state and it reads that symbol. This table is known as the *transition function* of the Turing machine.

7.1.2 Turing machines: a formal definition

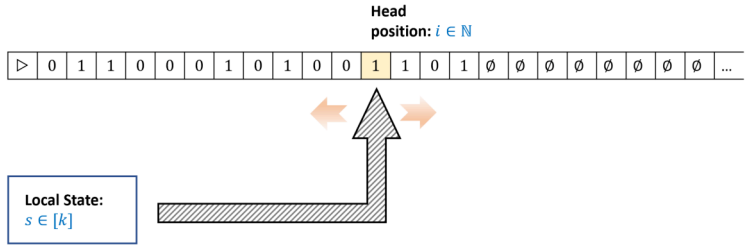


Figure 7.7: A Turing machine has access to a *tape* of unbounded length. At each point in the execution, the machine can read a single symbol of the tape, and based on that and its current state, write a new symbol, update the tape, decide whether to move left, right, stay, or halt.

Rules / transition function: $M: [k] \times \Sigma \to [k] \times \Sigma \times \{\mathbf{L}, \mathbf{R}, \mathbf{S}, \mathbf{H}\}$

```
If read 0 and state is 17 then change state to 29, write 1, and move left.
If read ∅ and state is 23 then change state to 15, write 0, and move right.
If read ▷ and state is 8  then change state to 12, write ▷, and halt.
...
```

The formal definition of Turing machines is as follows:

Definition 7.1 — Turing Machine. A (one tape) *Turing machine* with k states and alphabet $\Sigma \supseteq \{0,1,\triangleright,\varnothing\}$ is represented by a *transition function* $\delta_M : [k] \times \Sigma \to [k] \times \Sigma \times \{\mathsf{L},\mathsf{R},\mathsf{S},\mathsf{H}\}$.

For every $x \in \{0,1\}^*$, the *output* of M on input x, denoted by $M(x)$, is the result of the following process:

- We initialize T to be the sequence $\triangleright, x_0, x_1, \ldots, x_{n-1}, \varnothing, \varnothing, \ldots$, where $n = |x|$. (That is, $T[0] = \triangleright$, $T[i+1] = x_i$ for $i \in [n]$, and $T[i] = \varnothing$ for $i > n$.)
- We also initialize $i = 0$ and $s = 0$.
- We then repeat the following process:
  1. Let $(s', \sigma', D) = \delta_M(s, T[i])$.
  2. Set $s \to s'$, $T[i] \to \sigma'$.
  3. If $D = \mathsf{R}$ then set $i \to i+1$, if $D = \mathsf{L}$ then set $i \to \max\{i-1, 0\}$. (If $D = \mathsf{S}$ then we keep i the same.)
  4. If $D = \mathsf{H}$, then halt.
- If the process above halts, then M's output, denoted by $M(x)$, is the string $y \in \{0,1\}^*$ obtained by concatenating all the symbols in $\{0,1\}$ in positions $T[0], \ldots, T[i]$ where $i+1$ is the first location in the tape containing $\varnothing$.
- If The Turing machine does not halt then we denote $M(x) = \bot$.

P

You should make sure you see why this formal definition corresponds to our informal description of a Turing machine. To get more intuition on Turing machines, you can explore some of the online available simulators such as Martin Ugarte's, Anthony Morphett's, or Paul Rendell's.

One should not confuse the *transition function* δ_M of a Turing machine M with the function that the machine computes. The transition function δ_M is a *finite* function, with $k|\Sigma|$ inputs and $4k|\Sigma|$ outputs. (Can you see why?) The machine can compute an *infinite* function F that takes as input a string $x \in \{0,1\}^*$ of arbitrary length and might also produce an arbitrary length string as output.

In our formal definition, we identified the machine M with its transition function δ_M since the transition function tells us everything we need to know about the Turing machine. However, this choice of representation is somewhat arbitrary, and is based on our convention

that the state space is always the numbers $\{0, \ldots, k-1\}$ with 0 as the starting state. Other texts use different conventions, and so their mathematical definition of a Turing machine might look superficially different. However, these definitions describe the same computational process and have the same computational powers. Hence they are equivalent despite their superficial differences. See Section 7.7 for a comparison between Definition 7.1 and the way Turing Machines are defined in texts such as Sipser [Sip97].

7.1.3 Computable functions

We now turn to make one of the most important definitions in this book: *computable functions*.

Definition 7.2 — Computable functions. Let $F : \{0,1\}^* \to \{0,1\}^*$ be a (total) function and let M be a Turing machine. We say that M *computes* F if for every $x \in \{0,1\}^*$, $M(x) = F(x)$.

We say that a function F is *computable* if there exists a Turing machine M that computes it.

Defining a function "computable" if and only if it can be computed by a Turing machine might seem "reckless" but, as we'll see in Chapter 8, being computable in the sense of Definition 7.2 is equivalent to being computable in virtually any reasonable model of computation. This statement is known as the *Church-Turing Thesis*. (Unlike the *extended* Church-Turing Thesis which we discussed in Section 5.6, the Church-Turing thesis itself is widely believed and there are no candidate devices that attack it.)

Big Idea 9 We can precisely define what it means for a function to be computable by *any possible algorithm*.

This is a good point to remind the reader that *functions* are *not* the same as *programs*:

$$\text{Functions} \neq \text{Programs} .$$

A Turing machine (or program) M can *compute* some function F, but it is not the same as F. In particular, there can be more than one program to compute the same function. Being computable is a property of *functions*, not of machines.

We will often pay special attention to functions $F : \{0,1\}^* \to \{0,1\}$ that have a single bit of output. Hence we give a special name for the set of computable functions of this form.

Definition 7.3 — The class R. We define **R** be the set of all *computable* functions $F : \{0,1\}^* \to \{0,1\}$.

Remark 7.4 — Functions vs. languages. As discussed in Section 6.1.2, many texts use the terminology of "languages" rather than functions to refer to computational tasks. A Turing machine M *decides* a language L if for every input $x \in \{0,1\}^*$, $M(x)$ outputs 1 if and only if $x \in L$. This is equivalent to computing the Boolean function $F : \{0,1\}^* \to \{0,1\}$ defined as $F(x) = 1$ iff $x \in L$. A language L is *decidable* if there is a Turing machine M that decides it. For historical reasons, some texts also call such languages *recursive* , which is the reason that the letter **R** is often used to denote the set of computable Boolean functions / decidable languages defined in Definition 7.3.

In this book we stick to the terminology of *functions* rather than languages, but all definitions and results can be easily translated back and forth by using the equivalence between the function $F : \{0,1\}^* \to \{0,1\}$ and the language $L = \{x \in \{0,1\}^* \mid F(x) = 1\}$.

7.1.4 Infinite loops and partial functions

One crucial difference between circuits/straight-line programs and Turing machines is the following. Looking at a NAND-CIRC program P, we can always tell how many inputs and how many outputs P has by simply looking at the `X` and `Y` variables. Furthermore, we are guaranteed that if we invoke P on any input, then *some* output will be produced.

In contrast, given a Turing machine M, we cannot determine a priori the length of M's output. In fact, we don't even know if an output would be produced at all! For example, it is straightforward to come up with a Turing machine whose transition function never outputs `H` and hence never halts.

If a machine M fails to stop and produce an output on some input x, then it cannot compute any total function F, since clearly on input x, M will fail to output $F(x)$. However, M can still compute a *partial function*.[1]

For example, consider the partial function *DIV* that on input a pair (a,b) of natural numbers, outputs $\lceil a/b \rceil$ if $b > 0$, and is undefined otherwise. We can define a Turing machine M that computes *DIV* on input a, b by outputting the first $c = 0, 1, 2, \ldots$ such that $cb \geq a$. If $a > 0$ and $b = 0$ then the machine M will never halt, but this is OK, since *DIV* is undefined on such inputs. If $a = 0$ and $b = 0$, the machine M

[1] A *partial function* F from a set A to a set B is a function that is only defined on a *subset* of A, (see Section 1.4.3). We can also think of such a function as mapping A to $B \cup \{\perp\}$ where $\perp$ is a special "failure" symbol such that $F(a) = \perp$ indicates the function F is not defined on a.

will output 0, which is also OK, since we don't care about what the program outputs on inputs on which *DIV* is undefined. Formally, we define computability of partial functions as follows:

Definition 7.5 — Computable (partial or total) functions. Let F be either a total or partial function mapping $\{0,1\}^*$ to $\{0,1\}^*$ and let M be a Turing machine. We say that M *computes* F if for every $x \in \{0,1\}^*$ on which F is defined, $M(x) = F(x)$. We say that a (partial or total) function F is *computable* if there is a Turing machine that computes it.

Note that if F is a total function, then it is defined on every $x \in \{0,1\}^*$ and hence in this case, Definition 7.5 is identical to Definition 7.2.

R

Remark 7.6 — Bot symbol. We often use $\perp$ as our special "failure symbol". If a Turing machine M fails to halt on some input $x \in \{0,1\}^*$ then we denote this by $M(x) = \perp$. This *does not* mean that M outputs some encoding of the symbol $\perp$ but rather that M enters into an infinite loop when given x as input.

If a partial function F is undefined on x then we can also write $F(x) = \perp$. Therefore one might think that Definition 7.5 can be simplified to requiring that $M(x) = F(x)$ for every $x \in \{0,1\}^*$, which would imply that for every x, M halts on x if and only if F is defined on x. However, this is not the case: for a Turing machine M to compute a partial function F it is not *necessary* for M to enter an infinite loop on inputs x on which F is not defined. All that is needed is for M to output $F(x)$ on values of x on which F is defined: on other inputs it is OK for M to output an arbitrary value such as 0, 1, or anything else, or not to halt at all. To borrow a term from the `C` programming language, on inputs x on which F is not defined, what M does is "undefined behavior".

7.2 TURING MACHINES AS PROGRAMMING LANGUAGES

The name "Turing machine", with its "tape" and "head" evokes a physical object, while in contrast we think of a *program* as a piece of text. But we can think of a Turing machine as a program as well. For example, consider the Turing machine M of Section 7.1.1 that computes the function PAL such that $PAL(x) = 1$ iff x is a palindrome. We can also describe this machine as a *program* using the Python-like pseudocode of the form below

```
# Gets an array Tape initialized to
# [">", x_0 , x_1 , .... , x_(n-1), " ", " ", ...]
# At the end of the execution, Tape[1] is equal to 1
# if x is a palindrome and is equal to 0 otherwise
def PAL(Tape):
    head = 0
    state = 0 # START
    while (state != 12):
        if (state == 0 && Tape[head]=='0'):
            state = 3 # LOOK_FOR_0
            Tape[head] = 'x'
            head += 1 # move right
        if (state==0 && Tape[head]=='1')
            state = 4 # LOOK_FOR_1
            Tape[head] = 'x'
            head += 1 # move right
        ... # more if statements here
```

The precise details of this program are not important. What matters is that we can describe Turing machines as *programs*. Moreover, note that when translating a Turing machine into a program, the *tape* becomes a *list* or *array* that can hold values from the finite set Σ.[2] The *head position* can be thought of as an integer-valued variable that holds integers of unbounded size. The *state* is a *local register* that can hold one of a fixed number of values in $[k]$.

[2] Most programming languages use arrays of fixed size, while a Turing machine's tape is unbounded. But of course there is no need to store an infinite number of $\varnothing$ symbols. If you want, you can think of the tape as a list that starts off just long enough to store the input, but is dynamically grown in size as the Turing machine's head explores new positions.

More generally we can think of every Turing machine M as equivalent to a program similar to the following:

```
# Gets an array Tape initialized to
# [">", x_0 , x_1 , .... , x_(n-1), " ", " ", ...]
def M(Tape):
    state = 0
    i     = 0 # holds head location
    while (True):
        # Move head, modify state, write to tape
        # based on current state and cell at head
        # below are just examples for how program looks
         ↪  for a particular transition function
        if Tape[i]=="0" and state==7: #
         ↪  δ_M(7,"0")=(19,"1","R")
            Tape[i]="1"
            i += 1

            state = 19
```

```
elif Tape[i]==">" and state == 13: #
 ↪  δ_M(13,">")=(15,"0","S")
    Tape[i]="0"
    state = 15
elif ...
...
elif Tape[i]==">" and state == 29: #
 ↪  δ_M(29,">")=(.,.,"H")
    break # Halt
```

If we wanted to use only *Boolean* (i.e., $0/1$-valued) variables, then we can encode the `state` variables using $\lceil \log k \rceil$ bits. Similarly, we can represent each element of the alphabet Σ using $\ell = \lceil \log |\Sigma| \rceil$ bits and hence we can replace the Σ-valued array `Tape[]` with ℓ Boolean-valued arrays `Tape0[]`,..., `Tape`$(\ell-1)$`[]`.

7.2.1 The NAND-TM Programming language

We now introduce the *NAND-TM programming language*, which captures the power of a Turing machine with a programming-language formalism. Like the difference between Boolean circuits and Turing machines, the main difference between NAND-TM and NAND-CIRC is that NAND-TM models a *single uniform algorithm* that can compute a function that takes inputs of *arbitrary lengths*. To do so, we extend the NAND-CIRC programming language with two constructs:

- *Loops*: NAND-CIRC is a *straight-line* programming language- a NAND-CIRC program of s lines takes exactly s steps of computation and hence in particular, cannot even touch more than $3s$ variables. *Loops* allow us to use a fixed-length program to encode the instructions for a computation that can take an arbitrary amount of time.

- *Arrays*: A NAND-CIRC program of s lines touches at most $3s$ variables. While we can use variables with names such as `Foo_17` or `Bar[22]` in NAND-CIRC, they are not true arrays, since the number in the identifier is a constant that is "hardwired" into the program. NAND-TM contains actual arrays that can have a length that is not a priori bounded.

Thus a good way to remember NAND-TM is using the following informal equation:

$$\text{NAND-TM} = \text{NAND-CIRC} + \text{loops} + \text{arrays} \tag{7.1}$$

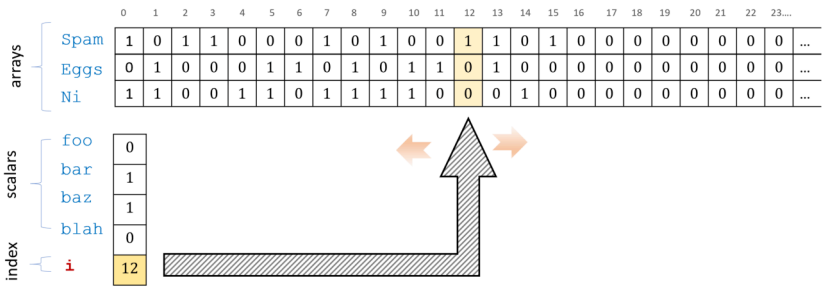


Figure 7.8: A NAND-TM program has *scalar* variables that can take a Boolean value, *array* variables that hold a sequence of Boolean values, and a special *index* variable `i` that can be used to index the array variables. We refer to the `i`-th value of the array variable `Spam` using `Spam[i]`. At each iteration of the program the index variable can be incremented or decremented by one step using the `MODANDJUMP` operation.

R

Remark 7.7 — NAND-CIRC + loops + arrays = everything.. As we will see, adding loops and arrays to NAND-CIRC is enough to capture the full power of all programming languages! Hence we could replace "NAND-TM" with any of *Python*, *C*, *Javascript*, *OCaml*, etc. in the left-hand side of (7.1). But we're getting ahead of ourselves: this issue will be discussed in Chapter 8.

Concretely, the NAND-TM programming language adds the following features on top of NAND-CIRC (see Fig. 7.8):

- We add a special *integer valued* variable `i`. All other variables in NAND-TM are *Boolean valued* (as in NAND-CIRC).
- Apart from `i` NAND-TM has two kinds of variables: *scalars* and *arrays*. *Scalar* variables hold one bit (just as in NAND-CIRC). *Array* variables hold an unbounded number of bits. At any point in the computation we can access the array variables at the location indexed by `i` using `Foo[i]`. We cannot access the arrays at locations other than the one pointed to by `i`.
- We use the convention that *arrays* always start with a capital letter, and *scalar variables* (which are never indexed with `i`) start with lowercase letters. Hence `Foo` is an array and `bar` is a scalar variable.
- The input and output `X` and `Y` are now considered *arrays* with values of zeroes and ones. (There are also two other special arrays `X_nonblank` and `Y_nonblank`, see below.)
- We add a special `MODANDJUMP` instruction that takes two Boolean variables a, b as input and does the following:
  - If $a = 1$ and $b = 1$ then `MODANDJUMP`(a, b) increments `i` by one and jumps to the first line of the program.

  - If $a = 0$ and $b = 1$ then `MODANDJUMP`(a, b) decrements `i` by one and jumps to the first line of the program. (If `i` is already equal to 0 then it stays at 0.)
  - If $a = 1$ and $b = 0$ then `MODANDJUMP`(a, b) jumps to the first line of the program without modifying `i`.
  - If $a = b = 0$ then `MODANDJUMP`(a, b) halts execution of the program.

- The `MODANDJUMP` instruction always appears in the last line of a NAND-TM program and nowhere else.

Default values. We need one more convention to handle "default values". Turing machines have the special symbol $\varnothing$ to indicate that tape location is "blank" or "uninitialized". In NAND-TM there is no such symbol, and all variables are *Boolean*, containing either 0 or 1. All variables and locations of arrays default to 0 if they have not been initialized to another value. To keep track of whether a 0 in an array corresponds to a true zero or to an uninitialized cell, a programmer can always add to an array `Foo` a "companion array" `Foo_nonblank` and set `Foo_nonblank[i]` to 1 whenever the `i` th location is initialized. In particular, we will use this convention for the input and output arrays `X` and `Y`. A NAND-TM program has *four* special arrays `X`, `X_nonblank`, `Y`, and `Y_nonblank`. When a NAND-TM program is executed on input $x \in \{0,1\}^*$ of length n, the first n cells of the array `X` are initialized to $x_0, \ldots, x_{n-1}$ and the first n cells of the array `X_nonblank` are initialized to 1. (All uninitialized cells default to 0.) The output of a NAND-TM program is the string `Y[0]`, ..., `Y[`$m-1$`]` where m is the smallest integer such that `Y_nonblank[`m`]`$= 0$. A NAND-TM program gets called with `X` and `X_nonblank` initialized to contain the input, and writes to `Y` and `Y_nonblank` to produce the output.

Formally, NAND-TM programs are defined as follows:

Definition 7.8 — NAND-TM programs. A *NAND-TM program* consists of a sequence of lines of the form `foo = NAND(bar,blah)` ending with a line of the form `MODANDJUMP(foo,bar)`, where `foo`,`bar`,`blah` are either *scalar variables* (sequences of letters, digits, and underscores) or *array variables* of the form `Foo[i]` (starting with capital letters and indexed by `i`). The program has the array variables `X`, `X_nonblank`, `Y`, `Y_nonblank` and the index variable `i` built in, and can use additional array and scalar variables.

If P is a NAND-TM program and $x \in \{0,1\}^*$ is an input then an execution of P on x is the following process:

1. The arrays `X` and `X_nonblank` are initialized by `X[`i`]=` x_i and `X_nonblank[`i`]=` 1 for all $i \in [|x|]$. All other variables and cells are initialized to 0. The index variable `i` is also initialized to 0.

2. The program is executed line by line. When the last line `MODAND-JUMP(foo,bar)` is executed we do as follows:

   a. If `foo=` 1 and `bar=` 0, jump to the first line without modifying the value of `i`.

   b. If `foo=` 1 and `bar=` 1, increment `i` by one and jump to the first line.

   c. If `foo=` 0 and `bar=` 1, decrement `i` by one (unless it is already zero) and jump to the first line.

   d. If `foo=` 0 and `bar=` 0, halt and output `Y[0]`, ..., `Y[`$m-1$`]` where m is the smallest integer such that `Y_nonblank[`m`]=` 0.

7.2.2 Sneak peak: NAND-TM vs Turing machines

As the name implies, NAND-TM programs are a direct implementation of Turing machines in programming language form. We will show the equivalence below, but you can already see how the components of Turing machines and NAND-TM programs correspond to one another:

Table 7.2: Turing Machine and NAND-TM analogs

Turing Machine	**NAND-TM program**
State: single register that takes values in $[k]$	*Scalar variables:* Several variables such as `foo`, `bar` etc.. each taking values in $\{0,1\}$.
Tape: One tape containing values in a finite set Σ. Potentially infinite but $T[t]$ defaults to $\varnothing$ for all locations t that have not been accessed.	*Arrays:* Several arrays such as `Foo`, `Bar` etc.. for each such array `Arr` and index j, the value of `Arr` at position j is either 0 or 1. The value defaults to 0 for position that have not been written to.
Head location: A number $i \in \mathbb{N}$ that encodes the position of the head.	*Index variable:* The variable `i` that can be used to access the arrays.

Turing Machine	NAND-TM program
Accessing memory: At every step the Turing machine has access to its local state, but can only access the tape at the position of the current head location.	*Accessing memory:* At every step a NAND-TM program has access to all the scalar variables, but can only access the arrays at the location `i` of the index variable
Control of location: In each step the machine can move the head location by at most one position.	*Control of index variable:* In each iteration of its main loop the program can modify the index `i` by at most one.

7.2.3 Examples

We now present some examples of NAND-TM programs.

■ **Example 7.9 — Increment in NAND-TM.** The following is a NAND-TM program to compute the *increment function*. That is, $INC : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for every $x \in \{0,1\}^n$, $INC(x)$ is the $n+1$ bit long string y such that if $X = \sum_{i=0}^{n-1} x_i \cdot 2^i$ is the number represented by x, then y is the (least-significant digit first) binary representation of the number $X+1$.

We start by describing the program using "syntactic sugar" for NAND-CIRC for the `IF`, `XOR` and `AND` functions (as well as the constant `one` function, and the function `COPY` that just maps a bit to itself).

```
carry = IF(started,carry,one(started))
started = one(started)
Y[i] = XOR(X[i],carry)
carry = AND(X[i],carry)
Y_nonblank[i] = one(started)
MODANDJUMP(X_nonblank[i],X_nonblank[i])
```

Since we used syntactic sugar, the above is not, strictly speaking, a valid NAND-TM program. However, by "opening up" all the syntactic sugar, we get the following "sugar free" valid program to compute the same function.

```
temp_0 = NAND(started,started)
temp_1 = NAND(started,temp_0)
temp_2 = NAND(started,started)
temp_3 = NAND(temp_1,temp_2)
```

```
temp_4 = NAND(carry,started)
carry = NAND(temp_3,temp_4)
temp_6 = NAND(started,started)
started = NAND(started,temp_6)
temp_8 = NAND(X[i],carry)
temp_9 = NAND(X[i],temp_8)
temp_10 = NAND(carry,temp_8)
Y[i] = NAND(temp_9,temp_10)
temp_12 = NAND(X[i],carry)
carry = NAND(temp_12,temp_12)
temp_14 = NAND(started,started)
Y_nonblank[i] = NAND(started,temp_14)
MODANDJUMP(X_nonblank[i],X_nonblank[i])
```

■ **Example 7.10 — XOR in NAND-TM.** The following is a NAND-TM program to compute the XOR function on inputs of arbitrary length. That is $XOR : \{0,1\}^* \rightarrow \{0,1\}$ such that $XOR(x) = \sum_{i=0}^{|x|-1} x_i \mod 2$ for every $x \in \{0,1\}^*$. Once again, we use a certain "syntactic sugar". Specifically, we access the arrays `X` and `Y` at their zero-th entry, while NAND-TM only allows access to arrays in the coordinate of the variable `i`.

```
temp_0 = NAND(X[0],X[0])
Y_nonblank[0] = NAND(X[0],temp_0)
temp_2 = NAND(X[i],Y[0])
temp_3 = NAND(X[i],temp_2)
temp_4 = NAND(Y[0],temp_2)
Y[0] = NAND(temp_3,temp_4)
MODANDJUMP(X_nonblank[i],X_nonblank[i])
```

To transform the program above to a valid NAND-TM program, we can transform references such as `X[0]` and `Y[0]` to scalar variables `x_0` and `y_0` (similarly we can transform any reference of the form `Foo[17]` or `Bar[15]` to scalars such as `foo_17` and `bar_15`). We then need to add code to load the value of `X[0]` to `x_0` and similarly to write to `Y[0]` the value of `y_0`, but this is not hard to do. Using the fact that variables are initialized to zero by default, we can create a variable `init` which will be set to 1 at the end of the first iteration and not changed since then. We can then add an array `Atzero` and code that will modify `Atzero[i]` to 1 if `init` is 0 and otherwise leave it as it is. This will ensure that `Atzero[i]` is equal to 1 if and only if `i` is set to zero, and allow the program to know when we are at the zeroth location. Thus we can add code

to read and write to the corresponding scalars `x_0`, `y_0` when we are at the zeroth location, and also code to move `i` to zero and then halt at the end. Working this out fully is somewhat tedious, but can be a good exercise.

Working out the above two examples can go a long way towards understanding the NAND-TM language. See our GitHub repository for a full specification of the NAND-TM language.

7.3 EQUIVALENCE OF TURING MACHINES AND NAND-TM PROGRAMS

Given the above discussion, it might not be surprising that Turing machines turn out to be equivalent to NAND-TM programs. Indeed, we designed the NAND-TM language to have this property. Nevertheless, this is a significant result, and the first of many other such equivalence results we will see in this book.

Theorem 7.11 — Turing machines and NAND-TM programs are equivalent. For every $F : \{0,1\}^* \rightarrow \{0,1\}^*$, F is computable by a NAND-TM program P if and only if there is a Turing machine M that computes F.

Proof Idea:

To prove such an equivalence theorem, we need to show two directions. We need to be able to **(1)** transform a Turing machine M to a NAND-TM program P that computes the same function as M and **(2)** transform a NAND-TM program P into a Turing machine M that computes the same function as P.

The idea of the proof is illustrated in Fig. 7.9. To show **(1)**, given a Turing machine M, we will create a NAND-TM program P that will have an array `Tape` for the tape of M and scalar (i.e., non-array) variable(s) `state` for the state of M. Specifically, since the state of a Turing machine is not in $\{0,1\}$ but rather in a larger set $[k]$, we will use $\lceil \log k \rceil$ variables `state_0` , ..., `state_`$\lceil \log k \rceil - 1$ variables to store the representation of the state. Similarly, to encode the larger alphabet Σ of the tape, we will use $\lceil \log |\Sigma| \rceil$ arrays `Tape_0` , ..., `Tape_`$\lceil \log |\Sigma| \rceil - 1$, such that the ith location of these arrays encodes the ith symbol in the tape for every tape. Using the fact that *every* function can be computed by a NAND-CIRC program, we will be able to compute the transition function of M, replacing moving left and right by decrementing and incrementing `i` respectively.

We show **(2)** using very similar ideas. Given a program P that uses a array variables and b scalar variables, we will create a Turing machine with about 2^b states to encode the values of scalar variables, and an alphabet of about 2^a so we can encode the arrays using our tape. (The reason the sizes are only "about" 2^a and 2^b is that we need to add some symbols and steps for bookkeeping purposes.) The Turing machine M simulates each iteration of the program P by updating its state and tape accordingly.

⋆

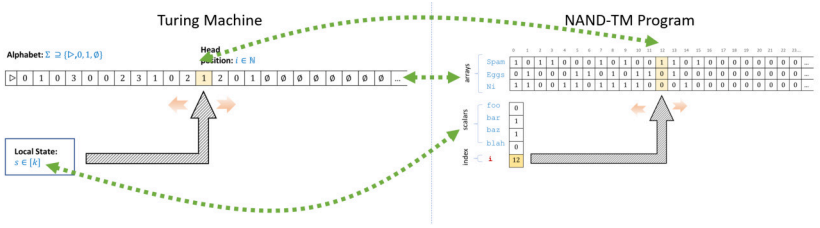


Figure 7.9: Comparing a Turing machine to a NAND-TM program. Both have an unbounded memory component (the *tape* for a Turing machine, and the *arrays* for a NAND-TM program), as well as a constant local memory (*state* for a Turing machine, and *scalar variables* for a NAND-TM program). Both can only access at each step one location of the unbounded memory, this is the "head" location for a Turing machine, and the value of the index variable `i` for a NAND-TM program.

Proof of Theorem 7.11. We start by proving the "if" direction of Theorem 7.11. Namely we show that given a Turing machine M, we can find a NAND-TM program P_M such that for every input x, if M halts on input x with output y then $P_M(x) = y$. Since our goal is just to show such a program P_M *exists*, we don't need to write out the full code of P_M line by line, and can take advantage of our various "syntactic sugar" in describing it.

The key observation is that by Theorem 4.12 we can compute *every* finite function using a NAND-CIRC program. In particular, consider the transition function $\delta_M : [k] \times \Sigma \to [k] \times \Sigma \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}, \mathsf{H}\}$ of our Turing machine. We can encode its components as follows:

- We encode $[k]$ using $\{0,1\}^\ell$ and Σ using $\{0,1\}^{\ell'}$, where $\ell = \lceil \log k \rceil$ and $\ell' = \lceil \log |\Sigma| \rceil$.
- We encode the set $\{\mathsf{L}, \mathsf{R}, \mathsf{S}, \mathsf{H}\}$ using $\{0,1\}^2$. We will choose the encoding $\mathsf{L} \mapsto 01$, $\mathsf{R} \mapsto 11$, $\mathsf{S} \mapsto 10$, $\mathsf{H} \mapsto 00$. (This conveniently corresponds to the semantics of the `MODANDJUMP` operation.)

Hence we can identify δ_M with a function $\overline{M} : \{0,1\}^{\ell+\ell'} \to \{0,1\}^{\ell+\ell'+2}$, mapping strings of length $\ell + \ell'$ to strings of length $\ell + \ell' + 2$. By Theorem 4.12 there exists a finite length NAND-CIRC program `ComputeM` that computes this function $\overline{M}$. The idea behind the NAND-TM program to simulate M is to:

1. Use variables `state_0` ... `state_`$\ell - 1$ to encode M's state.

2. Use arrays `Tape_`0`[]` ... `Tape_`$\ell' - 1$`[]` to encode M's tape.

3. Use the fact that transition is finite and computable by NAND-CIRC program.

Given the above, we can write code of the form:

`state_`0 ... `state_`$\ell - 1$, `Tape_`0`[i]`... `Tape_`$\ell' - 1$`[i]`, `dir0`,`dir1` $\leftarrow$ `TRANSITION(` `state_`0 ... `state_`$\ell - 1$, `Tape_`0`[i]`... `Tape_`$\ell' - 1$`[i]`, `dir0`,`dir1` `)`

`MODANDJUMP(dir0,dir1)`

Every step of the main loop of the above program perfectly mimics the computation of the Turing machine M, and so the program carries out exactly the definition of computation by a Turing machine as per Definition 7.1.

For the other direction, suppose that P is a NAND-TM program with s lines, ℓ scalar variables, and ℓ' array variables. We will show that there exists a Turing machine M_P with $2^\ell + C$ states and alphabet Σ of size $C' + 2^{\ell'}$ that computes the same functions as P (where C, C' are some constants to be determined later).

Specifically, consider the function $\overline{P} : \{0,1\}^\ell \times \{0,1\}^{\ell'} \rightarrow \{0,1\}^\ell \times \{0,1\}^{\ell'}$ that on input the contents of P's scalar variables and the contents of the array variables at location `i` in the beginning of an iteration, outputs all the new values of these variables at the last line of the iteration, right before the `MODANDJUMP` instruction is executed.

If `foo` and `bar` are the two variables that are used as input to the `MODANDJUMP` instruction, then based on the values of these variables we can compute whether `i` will increase, decrease or stay the same, and whether the program will halt or jump back to the beginning. Hence a Turing machine can simulate an execution of P in one iteration using a finite function applied to its alphabet. The overall operation of the Turing machine will be as follows:

1. The machine M_P encodes the contents of the array variables of P in its tape and the contents of the scalar variables in (part of) its state. Specifically, if P has ℓ local variables and t arrays, then the state space of M will be large enough to encode all 2^ℓ assignments to the local variables, and the alphabet Σ of M will be large enough to encode all 2^t assignments for the array variables at each location. The head location corresponds to the index variable `i`.

2. Recall that every line of the program P corresponds to reading and writing either a scalar variable, or an array variable at the location `i`. In one iteration of P the value of `i` remains fixed, and so the machine M can simulate this iteration by reading the values of all array variables at `i` (which are encoded by the single symbol in the alphabet Σ located at the `i`-th cell of the tape) , reading the values

of all scalar variables (which are encoded by the state), and updating both. The transition function of M can output $\mathsf{L}, \mathsf{S}, \mathsf{R}$ depending on whether the values given to the `MODANDJUMP` operation are 01, 10 or 11 respectively.

3. When the program halts (i.e., `MODANDJUMP` gets 00) then the Turing machine will enter into a special loop to copy the results of the `Y` array into the output and then halt. We can achieve this by adding a few more states.

The above is not a full formal description of a Turing machine, but our goal is just to show that such a machine exists. One can see that M_P simulates every step of P, and hence computes the same function as P.

■

Remark 7.12 — Running time equivalence (optional). If we examine the proof of Theorem 7.11 then we can see that every iteration of the loop of a NAND-TM program corresponds to one step in the execution of the Turing machine. We will come back to this question of measuring the number of computation steps later in this course. For now, the main take away point is that NAND-TM programs and Turing machines are essentially equivalent in power even when taking running time into account.

7.3.1 Specification vs implementation (again)

Once you understand the definitions of both NAND-TM programs and Turing machines, Theorem 7.11 is straightforward. Indeed, NAND-TM programs are not as much a different model from Turing machines as they are simply a reformulation of the same model using programming language notation. You can think of the difference between a Turing machine and a NAND-TM program as the difference between representing a number using decimal or binary notation. In contrast, the difference between a *function* F and a Turing machine that computes F is much more profound: it is like the difference between the equation $x^2 + x = 12$, and the number 3 that is a solution for this equation. For this reason, while we take special care in distinguishing *functions* from *programs* or *machines*, we will often identify the two latter concepts. We will move freely between describing an algorithm as a Turing machine or as a NAND-TM program (as well as some of the other equivalent computational models we will see in Chapter 8 and beyond).

Table 7.3: Specification vs Implementation formalisms

Setting	*Specification*	*Implementation*
Finite computation	**Functions** mapping $\{0,1\}^n$ to $\{0,1\}^m$	**Circuits**, **Straightline programs**
Infinite computation	**Functions** mapping $\{0,1\}^*$ to $\{0,1\}$ or to $\{0,1\}^*$.	**Algorithms**, **Turing Machines**, **Programs**

7.4 NAND-TM SYNTACTIC SUGAR

Just like we did with NAND-CIRC in Chapter 4, we can use "syntactic sugar" to make NAND-TM programs easier to write. For starters, we can use all of the syntactic sugar of NAND-CIRC, such as macro definitions and conditionals (i.e., if/then). However, we can go beyond this and achieve (for example):

- Inner loops such as the `while` and `for` operations common to many programming languages.
- Multiple index variables (e.g., not just `i` but we can add `j`, `k`, etc.).
- Arrays with more than one dimension (e.g., `Foo[i][j]`, `Bar[i][j][k]` etc.)

In all of these cases (and many others) we can implement the new feature as mere "syntactic sugar" on top of standard NAND-TM. This means that the set of functions computable by NAND-TM with this feature is the same as the set of functions computable by standard NAND-TM. Similarly, we can show that the set of functions computable by Turing machines that have more than one tape, or tapes of more dimensions than one, is the same as the set of functions computable by standard Turing machines.

7.4.1 "GOTO" and inner loops

We can implement more advanced *looping constructs* than the simple `MODANDJUMP`. For example, we can implement `GOTO`. A `GOTO` statement corresponds to jumping to a specific line in the execution. For example, if we have code of the form

```
"start":  do foo
   GOTO("end")
"skip": do bar
"end": do blah
```

then the program will only do `foo` and `blah` as when it reaches the line `GOTO("end")` it will jump to the line labeled with `"end"`. We can achieve the effect of `GOTO` in NAND-TM using conditionals. In the code below, we assume that we have a variable `pc` that can take strings of some constant length. This can be encoded using a finite number of Boolean variables `pc_0`, `pc_1`, ..., `pc_`$k-1$, and so when we write below `pc = "label"` what we mean is something like `pc_0 = 0`,`pc_1 = 1`, ... (where the bits $0, 1, \ldots$ correspond to the encoding of the finite string `"label"` as a string of length k). We also assume that we have access to conditional (i.e., `if` statements), which we can emulate using syntactic sugar in the same way as we did in NAND-CIRC.

To emulate a GOTO statement, we will first modify a program P of the form

```
do foo
do bar
do blah
```

to have the following form (using syntactic sugar for `if`):

```
pc = "line1"
if (pc=="line1"):
    do foo
    pc = "line2"
if (pc=="line2"):
    do bar
    pc = "line3"
if (pc=="line3"):
    do blah
```

These two programs do the same thing. The variable `pc` corresponds to the "program counter" and tells the program which line to execute next. We can see that if we wanted to emulate a `GOTO("line3")` then we could simply modify the instruction `pc = "line2"` to be `pc = "line3"`.

In NAND-CIRC we could only have `GOTO`s that go forward in the code, but since in NAND-TM everything is encompassed within a large outer loop, we can use the same ideas to implement `GOTO`s that can go backward, as well as conditional loops.

Other loops. Once we have `GOTO`, we can emulate all the standard loop constructs such as `while`, `do .. until` or `for` in NAND-TM as well. For example, we can replace the code

```
while foo:
    do blah
do bar
```

with

```
"loop":
    if NOT(foo): GOTO("next")
    do blah
    GOTO("loop")
"next":
    do bar
```

Remark 7.13 — GOTO's in programming languages. The `GOTO` statement was a staple of most early programming languages, but has largely fallen out of favor and is not included in many modern languages such as *Python*, *Java*, *Javascript*. In 1968, Edsger Dijsktra wrote a famous letter titled "Go to statement considered harmful." (see also Fig. 7.10). The main trouble with `GOTO` is that it makes analysis of programs more difficult by making it harder to argue about *invariants* of the program.

When a program contains a loop of the form:

```
for j in range(100):
    do something


do blah
```

you know that the line of code `do blah` can only be reached if the loop ended, in which case you know that `j` is equal to 100, and might also be able to argue other properties of the state of the program. In contrast, if the program might jump to `do blah` from any other point in the code, then it's very hard for you as the programmer to know what you can rely upon in this code. As Dijkstra said, such invariants are important because _ "our intellectual powers are rather geared to master static relations and .. our powers to visualize processes evolving in time are relatively poorly developed"_ and so *" we should ... do ...our utmost best to shorten the conceptual gap between the static program and the dynamic process."*

That said, `GOTO` is still a major part of lower level languages where it is used to implement higher-level looping constructs such as `while` and `for` loops. For example, even though *Java* doesn't have a `GOTO` statement, the Java Bytecode (which is a lower-level representation of Java) does have such a statement. Similarly, Python bytecode has instructions such as `POP_JUMP_IF_TRUE` that implement the `GOTO` functionality, and similar instructions are included in many assembly languages. The way we use `GOTO` to implement a higher-level functionality in NAND-TM is

reminiscent of the way these various jump instructions are used to implement higher-level looping constructs.

Figure 7.10: XKCD's take on the `GOTO` statement.

7.5 UNIFORMITY, AND NAND VS NAND-TM (DISCUSSION)

While NAND-TM adds extra operations over NAND-CIRC, it is not exactly accurate to say that NAND-TM programs or Turing machines are "more powerful" than NAND-CIRC programs or Boolean circuits. NAND-CIRC programs, having no loops, are simply not applicable for computing functions with an unbounded number of inputs. Thus, to compute a function $F : \{0,1\}^* \to \{0,1\}^*$ using NAND-CIRC (or equivalently, Boolean circuits) we need a *collection* of programs/circuits: one for every input length.

The key difference between NAND-CIRC and NAND-TM is that NAND-TM allows us to express the fact that the algorithm for computing parities of length-100 strings is really the same one as the algorithm for computing parities of length-5 strings (or similarly the fact that the algorithm for adding n-bit numbers is the same for every n, etc.). That is, one can think of the NAND-TM program for general parity as the "seed" out of which we can grow NAND-CIRC programs for length 10, length 100, or length 1000 parities as needed.

This notion of a single algorithm that can compute functions of all input lengths is known as *uniformity* of computation. Hence we think of Turing machines / NAND-TM as *uniform* models of computation, as opposed to Boolean circuits or NAND-CIRC, which are *non-uniform* models, in which we have to specify a different program for every input length.

Looking ahead, we will see that this uniformity leads to another crucial difference between Turing machines and circuits. Turing machines can have inputs and outputs that are longer than the description of the machine as a string, and in particular there exists a Turing machine that can "self replicate" in the sense that it can print its own code. The notion of "self replication", and the related notion of "self reference" are crucial to many aspects of computation, and beyond that to life itself, whether in the form of digital or biological programs.

For now, what you ought to remember is the following differences between *uniform* and *non-uniform* computational models:

- **Non-uniform computational models:** Examples are *NAND-CIRC programs* and *Boolean circuits*. These are models where each individual program/circuit can compute a *finite* function $f : \{0,1\}^n \to \{0,1\}^m$. We have seen that *every* finite function can be computed by *some* program/circuit. To discuss computation of an *infinite* function $F : \{0,1\}^* \to \{0,1\}^*$ we need to allow a *sequence* $\{P_n\}_{n \in \mathbb{N}}$ of

programs/circuits (one for every input length), but this does not capture the notion of a *single algorithm* to compute the function F.

- **Uniform computational models:** Examples are *Turing machines* and *NAND-TM programs*. These are models where a single program/-machine can take inputs of *arbitrary length* and hence compute an *infinite* function $F : \{0,1\}^* \rightarrow \{0,1\}^*$. The number of steps that a program/machine takes on some input is not a priori bounded in advance and in particular there is a chance that it will enter into an *infinite loop*. Unlike the non-uniform case, we have *not* shown that every infinite function can be computed by some NAND-TM program/Turing machine. We will come back to this point in Chapter 9.

Chapter Recap

- *Turing machines* capture the notion of a single algorithm that can evaluate functions of every input length.
- They are equivalent to *NAND-TM programs*, which add loops and arrays to NAND-CIRC.
- Unlike NAND-CIRC or Boolean circuits, the number of steps that a Turing machine takes on a given input is not fixed in advance. In fact, a Turing machine or a NAND-TM program can enter into an *infinite loop* on certain inputs, and not halt at all.

7.6 EXERCISES

Exercise 7.1 — Explicit NAND TM programming. Produce the code of a (syntactic-sugar free) NAND-TM program P that computes the (unbounded input length) *Majority* function $Maj : \{0,1\}^* \rightarrow \{0,1\}$ where for every $x \in \{0,1\}^*$, $Maj(x) = 1$ if and only if $\sum_{i=0}^{|x|} x_i > |x|/2$. We say "produce" rather than "write" because you do not have to write the code of P by hand, but rather can use the programming language of your choice to compute this code.

■

Exercise 7.2 — Computable functions examples. Prove that the following functions are computable. For all of these functions, you do not have to fully specify the Turing machine or the NAND-TM program that computes the function, but rather only prove that such a machine or program exists:

1. $INC : \{0,1\}^* \rightarrow \{0,1\}^*$ which takes as input a representation of a natural number n and outputs the representation of $n+1$.

2. $ADD : \{0,1\}^* \rightarrow \{0,1\}^*$ which takes as input a representation of a pair of natural numbers (n,m) and outputs the representation of $n+m$.

3. $MULT : \{0,1\}^* \rightarrow \{0,1\}^*$, which takes a representation of a pair of natural numbers (n,m) and outputs the representation of $n \dot{m}$.

4. $SORT : \{0,1\}^* \rightarrow \{0,1\}^*$ which takes as input the representation of a list of natural numbers $(a_0, \ldots, a_{n-1})$ and returns its sorted version $(b_0, \ldots, b_{n-1})$ such that for every $i \in [n]$ there is some $j \in [n]$ with $b_i = a_j$ and $b_0 \leq b_1 \leq \cdots \leq b_{n-1}$.

■

Exercise 7.3 — Two index NAND-TM. Define NAND-TM′ to be the variant of NAND-TM where there are *two* index variables `i` and `j`. Arrays can be indexed by either `i` or `j`. The operation `MODANDJUMP` takes four variables a, b, c, d and uses the values of c, d to decide whether to increment `j`, decrement `j` or keep it in the same value (corresponding to 01, 10, and 00 respectively). Prove that for every function $F : \{0,1\}^* \rightarrow \{0,1\}^*$, F is computable by a NAND-TM program if and only if F is computable by a NAND-TM′ program.

■

Exercise 7.4 — Two tape Turing machines. Define a *two tape Turing machine* to be a Turing machine which has two separate tapes and two separate heads. At every step, the transition function gets as input the location of the cells in the two tapes, and can decide whether to move each head independently. Prove that for every function $F : \{0,1\}^* \rightarrow \{0,1\}^*$, F is computable by a standard Turing machine if and only if F is computable by a two-tape Turing machine.

■

Exercise 7.5 — Two dimensional arrays. Define NAND-TM″ to be the variant of NAND-TM where just like NAND-TM′ defined in Exercise 7.3 there are two index variables `i` and `j`, but now the arrays are *two dimensional* and so we index an array `Foo` by `Foo[i][j]`. Prove that for every function $F : \{0,1\}^* \rightarrow \{0,1\}^*$, F is computable by a NAND-TM program if and only if F is computable by a NAND-TM′′ program.

■

Exercise 7.6 — Two dimensional Turing machines. Define a *two-dimensional Turing machine* to be a Turing machine in which the tape is *two dimensional*. At every step the machine can move Up, Down, Left, Right, or Stay. Prove that for every function $F : \{0,1\}^* \rightarrow \{0,1\}^*$, F is computable by a standard Turing machine if and only if F is computable by a two-dimensional Turing machine.

■

Exercise 7.7 Prove the following closure properties of the set **R** defined in Definition 7.3:

1. If $F \in \mathbf{R}$ then the function $G(x) = 1 - F(x)$ is in **R**.
2. If $F, G \in \mathbf{R}$ then the function $H(x) = F(x) \vee G(x)$ is in **R**.
3. If $F \in \mathbf{R}$ then the function F^* in in **R** where F^* is defined as follows: $F^*(x) = 1$ iff there exist some strings $w_0, \ldots, w_{k-1}$ such that $x = w_0 w_1 \cdots w_{k-1}$ and $F(w_i) = 1$ for every $i \in [k]$.
4. If $F \in \mathbf{R}$ then the function

$$G(x) = \begin{cases} \exists_{y \in \{0,1\}^{|x|}} F(xy) = 1 & \\ 0 & \text{otherwise} \end{cases}$$

is in **R**.

■

Exercise 7.8 — Oblivious Turing Machines (challenging). Define a Turing machine M to be *oblivious* if its head movements are independent of its input. That is, we say that M is oblivious if there exists an infinite sequence $MOVE \in \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}^\infty$ such that for every $x \in \{0,1\}^*$, the movements of M when given input x (up until the point it halts, if such point exists) are given by $MOVE_0, MOVE_1, MOVE_2, \ldots$.

Prove that for every function $F : \{0,1\}^* \to \{0,1\}^*$, if F is computable then it is computable by an oblivious Turing machine. See footnote for hint.[3]

[3] You can use the sequence R, L,R, R, L, L, R,R,R, L, L, L, ...

■

Exercise 7.9 — Single vs multiple bit. Prove that for every $F : \{0,1\}^* \to \{0,1\}^*$, the function F is computable if and only if the following function $G : \{0,1\}^* \to \{0,1\}$ is computable, where G is defined as follows:

$$G(x, i, \sigma) = \begin{cases} F(x)_i & i < |F(x)|, \sigma = 0 \\ 1 & i < |F(x)|, \sigma = 1 \\ 0 & i \geq |F(x)| \end{cases}$$

■

Exercise 7.10 — Uncomputability via counting. Recall that **R** is the set of all total functions from $\{0,1\}^*$ to $\{0,1\}$ that are computable by a Turing machine (see Definition 7.3). Prove that **R** is *countable*. That is, prove that there exists a one-to-one map $DtN : \mathbf{R} \to \mathbb{N}$. You can use the equivalence between Turing machines and NAND-TM programs.

■

Exercise 7.11 — Not every function is computable. Prove that the set of *all* total functions from $\{0,1\}^* \to \{0,1\}$ is *not* countable. You can use the results of Section 2.4. (We will see an *explicit* uncomputable function in Chapter 9.)

7.7 BIBLIOGRAPHICAL NOTES

Augusta Ada Byron, countess of Lovelace (1815-1852) lived a short but turbulent life, though is today most well known for her collaboration with Charles Babbage (see [Ste87] for a biography). Ada took an immense interest in Babbage's *analytical engine*, which we mentioned in Chapter 3. In 1842-3, she translated from Italian a paper of Menabrea on the engine, adding copious notes (longer than the paper itself). The quote in the chapter's beginning is taken from Nota A in this text. Lovelace's notes contain several examples of *programs* for the analytical engine, and because of this she has been called "the world's first computer programmer" though it is not clear whether they were written by Lovelace or Babbage himself [Hol01]. Regardless, Ada was clearly one of very few people (perhaps the only one outside of Babbage himself) to fully appreciate how important and revolutionary the idea of mechanizing computation truly is.

The books of Shetterly [She16] and Sobel [Sob17] discuss the history of human computers (who were female, more often than not) and their important contributions to scientific discoveries in astronomy and space exploration.

Alan Turing was one of the intellectual giants of the 20th century. He was not only the first person to define the notion of computation, but also invented and used some of the world's earliest computational devices as part of the effort to break the *Enigma* cipher during World War II, saving millions of lives. Tragically, Turing committed suicide in 1954, following his conviction in 1952 for homosexual acts and a court-mandated hormonal treatment. In 2009, British prime minister Gordon Brown made an official public apology to Turing, and in 2013 Queen Elizabeth II granted Turing a posthumous pardon. Turing's life is the subject of a great book and a mediocre movie.

Sipser's text [Sip97] defines a Turing machine as a *seven tuple* consisting of the state space, input alphabet, tape alphabet, transition function, starting state, accepting state, and rejecting state. Superficially this looks like a very different definition than Definition 7.1 but it is simply a different representation of the same concept, just as a graph can be represented in either adjacency list or adjacency matrix form.

One difference is that Sipser considers a general set of states Q that is not necessarily of the form $Q = \{0, 1, 2, \ldots, k-1\}$ for some natural number $k > 0$. Sipser also restricts his attention to Turing machines that output only a single bit and therefore designates two special *halting states*: the "0 halting state" (often known as the *rejecting state*) and the other as the "1 halting state" (often known as the *accepting state*).

Thus instead of writing 0 or 1 on an output tape, the machine will enter into one of these states and halt. This again makes no difference to the computational power, though we prefer to consider the more general model of multi-bit outputs. (Sipser presents the basic task of a Turing machine as that of *deciding a language* as opposed to computing a function, but these are equivalent, see Remark 7.4.)

Sipser considers also functions with input in Σ^* for an arbitrary alphabet Σ (and hence distinguishes between the *input alphabet* which he denotes as Σ and the *tape alphabet* which he denotes as Γ), while we restrict attention to functions with binary strings as input. Again this is not a major issue, since we can always encode an element of Σ using a binary string of length $\log\lceil|\Sigma|\rceil$. Finally (and this is a very minor point) Sipser requires the machine to either move left or right in every step, without the Stay operation, though staying in place is very easy to emulate by simply moving right and then back left.

Another definition used in the literature is that a Turing machine M *recognizes* a language L if for every $x \in L$, $M(x) = 1$ and for every $x \notin L$, $M(x) \in \{0, \bot\}$. A language L is *recursively enumerable* if there exists a Turing machine M that recognizes it, and the set of all recursively enumerable languages is often denoted by **RE**. We will not use this terminology in this book.

One of the first programming-language formulations of Turing machines was given by Wang [Wan57]. Our formulation of NAND-TM is aimed at making the connection with circuits more direct, with the eventual goal of using it for the Cook-Levin Theorem, as well as results such as $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$ and $\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$. The website esolangs.org features a large variety of esoteric Turing-complete programming languages. One of the most famous of them is Brainf*ck.

Learning Objectives:

- **Learn about RAM machines and the λ calculus.**
- **Equivalence between these and other models and Turing machines.**
- **Cellular automata and configurations of Turing machines.**
- **Understand the Church-Turing thesis.**

8
Equivalent models of computation

> *"All problems in computer science can be solved by another level of indirection"*, attributed to David Wheeler.

> *"Because we shall later compute with expressions for functions, we need a distinction between functions and forms and a notation for expressing this distinction. This distinction and a notation for describing it, from which we deviate trivially, is given by Church."*, John McCarthy, 1960 (in paper describing the LISP programming language)

So far we have defined the notion of computing a function using Turing machines, which are not a close match to the way computation is done in practice. In this chapter we justify this choice by showing that the definition of computable functions will remain the same under a wide variety of computational models. This notion is known as *Turing completeness* or *Turing equivalence* and is one of the most fundamental facts of computer science. In fact, a widely believed claim known as the *Church-Turing Thesis* holds that *every* "reasonable" definition of computable function is equivalent to being computable by a Turing machine. We discuss the Church-Turing Thesis and the potential definitions of "reasonable" in Section 8.8.

Some of the main computational models we discuss in this chapter include:

- **RAM Machines:** Turing machines do not correspond to standard computing architectures that have *Random Access Memory* (*RAM*). The mathematical model of RAM machines is much closer to actual computers, but we will see that it is equivalent in power to Turing machines. We also discuss a programming language variant of RAM machines, which we call NAND-RAM. The equivalence of Turing machines and RAM machines enables demonstrating the *Turing Equivalence* of many popular programming languages, including all general-purpose languages used in practice such as C, Python, JavaScript, etc.

- **Cellular Automata:** Many natural and artificial systems can be modeled as collections of simple components, each evolving according to simple rules based on its state and the state of its immediate neighbors. One well-known such example is Conway's Game of Life. To prove that cellular automata are equivalent to Turing machines we introduce the tool of *configurations* of Turing machines. These have other applications, and in particular are used in Chapter 11 to prove *Gödel's Incompleteness Theorem*: a central result in mathematics.

- **λ calculus:** The λ calculus is a model for expressing computation that originates from the 1930's, though it is closely connected to functional programming languages widely used today. Showing the equivalence of λ calculus to Turing machines involves a beautiful technique to eliminate recursion known as the "Y Combinator".

This chapter: A non-mathy overview

In this chapter we study *equivalence between models*. Two computational models are *equivalent* (also known as *Turing equivalent*) if they can compute the same set of functions. For example, we have seen that Turing machines and NAND-TM programs are equivalent since we can transform every Turing machine into a NAND-TM program that computes the same function, and similarly can transform every NAND-TM program into a Turing machine that computes the same function.

In this chapter we show this extends far beyond Turing machines. The techniques we develop allow us to show that all general-purpose programming languages (i.e., Python, C, Java, etc.) are *Turing Complete*, in the sense that they can simulate Turing machines and hence compute all functions that can be computed by a TM. We will also show the other direction- Turing machines can be used to simulate a program in any of these languages and hence compute any function computable by them. This means that all these programming language are *Turing equivalent*: they are equivalent in power to Turing machines and to each other. This is a powerful principle, which underlies behind the vast reach of Computer Science. Moreover, it enables us to "have our cake and eat it too"- since all these models are equivalent, we can choose the model of our convenience for the task at hand.

To achieve this equivalence, we define a new computational model known as *RAM machines*. RAM Machines capture the

architecture of modern computers more closely than Turing machines, but are still computationally equivalent to Turing machines.
Finally, we will show that Turing equivalence extends far beyond traditional programming languages. We will see that *cellular automata* which are a mathematical model of extremely simple natural systems is also Turing equivalent, and also see the Turing equivalence of the λ calculus - a logical system for expressing functions that is the basis for *functional programming languages* such as Lisp, OCaml, and more.
See Fig. 8.1 for an overview of the results of this chapter.

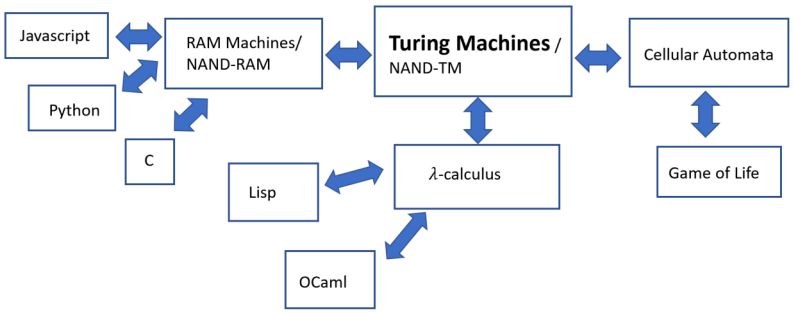


Figure 8.1: Some Turing-equivalent models. All of these are equivalent in power to Turing machines (or equivalently NAND-TM programs) in the sense that they can compute exactly the same class of functions. All of these are models for computing *infinite* functions that take inputs of unbounded length. In contrast, Boolean circuits / NAND-CIRC programs can only compute *finite* functions and hence are not Turing complete.

8.1 RAM MACHINES AND NAND-RAM

One of the limitations of Turing machines (and NAND-TM programs) is that we can only access one location of our arrays/tape at a time. If the head is at position 22 in the tape and we want to access the 957-th position then it will take us at least 923 steps to get there. In contrast, almost every programming language has a formalism for directly accessing memory locations. Actual physical computers also provide so called *Random Access Memory* (*RAM*) which can be thought of as a large array `Memory`, such that given an index p (i.e., memory address, or a *pointer*), we can read from and write to the pth location of `Memory`. ("Random access memory" is quite a misnomer since it has nothing to do with probability, but since it is a standard term in both the theory and practice of computing, we will use it as well.)

The computational model that models access to such a memory is the *RAM machine* (sometimes also known as the *Word RAM model*), as depicted in Fig. 8.2. The memory of a RAM machine is an array of unbounded size where each cell can store a single *word*, which we think of as a string in $\{0,1\}^w$ and also (equivalently) as a number in $[2^w]$. For example, many modern computing architectures

use 64 bit words, in which every memory location holds a string in $\{0,1\}^{64}$ which can also be thought of as a number between 0 and $2^{64}-1 = 18,446,744,073,709,551,615$. The parameter w is known as the *word size*. In practice often w is a fixed number such as 64, but when doing theory we model w as a parameter that can depend on the input length or number of steps. (You can think of 2^w as roughly corresponding to the largest memory address that we use in the computation.) In addition to the memory array, a RAM machine also contains a constant number of *registers* $r_0, \ldots, r_{k-1}$, each of which can also contain a single word.

The operations a RAM machine can carry out include:

- **Data movement:** Load data from a certain cell in memory into a register or store the contents of a register into a certain cell of memory. A RAM machine can directly access any cell of memory without having to move the "head" (as Turing machines do) to that location. That is, in one step a RAM machine can load into register r_i the contents of the memory cell indexed by register r_j, or store into the memory cell indexed by register r_j the contents of register r_i.
- **Computation:** RAM machines can carry out computation on registers such as arithmetic operations, logical operations, and comparisons.
- **Control flow:** As in the case of Turing machines, the choice of what instruction to perform next can depend on the state of the RAM machine, which is captured by the contents of its register.

We will not give a formal definition of RAM Machines, though the bibliographical notes section (Section 8.10) contains sources for such definitions. Just as the NAND-TM programming language models Turing machines, we can also define a *NAND-RAM programming language* that models RAM machines. The NAND-RAM programming language extends NAND-TM by adding the following features:

- The variables of NAND-RAM are allowed to be (non-negative) *integer valued* rather than only Boolean as is the case in NAND-TM. That is, a scalar variable `foo` holds a non-negative integer in $\mathbb{N}$ (rather than only a bit in $\{0,1\}$), and an array variable `Bar` holds an array of integers. As in the case of RAM machines, we will not allow integers of unbounded size. Concretely, each variable holds a number between 0 and $T-1$, where T is the number of steps that have been executed by the program so far. (You can ignore this restriction for now: if we want to hold larger numbers, we can simply execute dummy instructions; it will be useful in later chapters.)

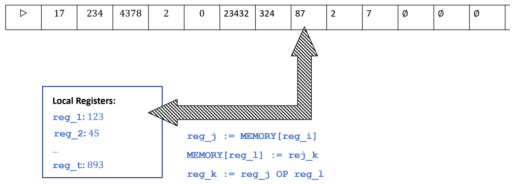


Figure 8.2: A *RAM Machine* contains a finite number of local registers, each of which holds an integer, and an unbounded memory array. It can perform arithmetic operations on its register as well as load to a register r the contents of the memory at the address indexed by the number in register r'.

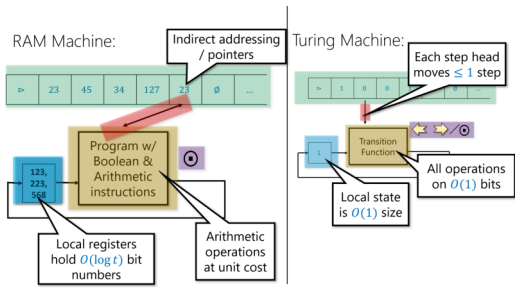


Figure 8.3: Different aspects of RAM machines and Turing machines. RAM machines can store integers in their local registers, and can read and write to their memory at a location specified by a register. In contrast, Turing machines can only access their memory in the head location, which moves at most one position to the right or left in each step.

- We allow *indexed access* to arrays. If `foo` is a scalar and `Bar` is an array, then `Bar[foo]` refers to the location of `Bar` indexed by the value of `foo`. (Note that this means we don't need to have a special index variable `i` anymore.)

- As is often the case in programming languages, we will assume that for Boolean operations such as `NAND`, a zero valued integer is considered as *false*, and a non-zero valued integer is considered as *true*.

- In addition to `NAND`, NAND-RAM also includes all the basic arithmetic operations of addition, subtraction, multiplication, (integer) division, as well as comparisons (equal, greater than, less than, etc..).

- NAND-RAM includes conditional statements `if/then` as part of the language.

- NAND-RAM contains looping constructs such as `while` and `do` as part of the language.

A full description of the NAND-RAM programming language is in the appendix. However, the most important fact you need to know about NAND-RAM is that you actually don't need to know much about NAND-RAM at all, since it is equivalent in power to Turing machines:

Theorem 8.1 — Turing Machines (aka NAND-TM programs) and RAM machines (aka NAND-RAM programs) are equivalent. For every function $F : \{0,1\}^* \to \{0,1\}^*$, F is computable by a NAND-TM program if and only if F is computable by a NAND-RAM program.

Since NAND-TM programs are equivalent to Turing machines, and NAND-RAM programs are equivalent to RAM machines, Theorem 8.1 shows that all these four models are equivalent to one another.

Proof Idea:

Clearly NAND-RAM is only more powerful than NAND-TM, and so if a function F is computable by a NAND-TM program then it can be computed by a NAND-RAM program. The challenging direction is to transform a NAND-RAM program P to an equivalent NAND-TM program Q. To describe the proof in full we will need to cover the full formal specification of the NAND-RAM language, and show how we can implement every one of its features as syntactic sugar on top of NAND-TM.

This can be done but going over all the operations in detail is rather tedious. Hence we will focus on describing the main ideas behind this

Figure 8.4: Overview of the steps in the proof of Theorem 8.1 simulating NANDRAM with NANDTM. We first use the inner loop syntactic sugar of Section 7.4.1 to enable loading an integer from an array to the index variable `i` of NANDTM. Once we can do that, we can simulate *indexed access* in NANDTM. We then use an embedding of $\mathbb{N}^2$ in $\mathbb{N}$ to simulate two dimensional bit arrays in NANDTM. Finally, we use the binary representation to encode one-dimensional arrays of integers as two dimensional arrays of bits hence completing the simulation of NANDRAM with NANDTM.

transformation. (See also Fig. 8.4.) NAND-RAM generalizes NAND-TM in two main ways: **(a)** adding *indexed access* to the arrays (ie.., `Foo[bar]` syntax) and **(b)** moving from *Boolean valued* variables to *integer valued* ones. The transformation has two steps:

1. *Indexed access of bit arrays:* We start by showing how to handle **(a)**. Namely, we show how we can implement in NAND-TM the operation `Setindex(Bar)` such that if `Bar` is an array that encodes some integer j, then after executing `Setindex(Bar)` the value of `i` will equal to j. This will allow us to simulate syntax of the form `Foo[Bar]` by `Setindex(Bar)` followed by `Foo[i]`.

2. *Two dimensional bit arrays:* We then show how we can use "syntactic sugar" to augment NAND-TM with *two dimensional arrays*. That is, have *two indices* `i` and `j` and *two dimensional arrays*, such that we can use the syntax `Foo[i][j]` to access the `(i,j)`-th location of `Foo`.

3. *Arrays of integers:* Finally we will encode a one dimensional array `Arr` of *integers* by a two dimensional `Arrbin` of *bits*. The idea is simple: if $a_{i,0}, \ldots, a_{i,\ell}$ is a binary (prefix-free) representation of `Arr[`i`]`, then `Arrbin[`i`][`j`]` will be equal to $a_{i,j}$.

Once we have arrays of integers, we can use our usual syntactic sugar for functions, `GOTO` etc. to implement the arithmetic and control flow operations of NAND-RAM.

⋆

The above approach is not the only way to obtain a proof of Theorem 8.1, see for example Exercise 8.1

R

Remark 8.2 — RAM machines / NAND-RAM and assembly language (optional). RAM machines correspond quite closely to actual microprocessors such as those in the Intel x86 series that also contains a large *primary memory* and a constant number of small registers. This is of course no accident: RAM machines aim at modeling more closely than Turing machines the architecture of actual computing systems, which largely follows the so called von Neumann architecture as described in the report [Neu45]. As a result, NAND-RAM is similar in its general outline to assembly languages such as x86 or NIPS. These assembly languages all have instructions to **(1)** move data from registers to memory, **(2)** perform arithmetic or logical computations on registers, and **(3)** conditional execution and loops ("if" and "goto", commonly known as "branches" and "jumps" in the context of assembly languages).
The main difference between RAM machines and actual microprocessors (and correspondingly between

NAND-RAM and assembly languages) is that actual microprocessors have a fixed word size w so that all registers and memory cells hold numbers in $[2^w]$ (or equivalently strings in $\{0,1\}^w$). This number w can vary among different processors, but common values are either 32 or 64. As a theoretical model, RAM machines do not have this limitation, but we rather let w be the logarithm of our running time (which roughly corresponds to its value in practice as well). Actual microprocessors also have a fixed number of registers (e.g., 14 general purpose registers in x86-64) but this does not make a big difference with RAM machines. It can be shown that RAM machines with as few as two registers are as powerful as full-fledged RAM machines that have an arbitrarily large constant number of registers.

Of course actual microprocessors have many features not shared with RAM machines as well, including parallelism, memory hierarchies, and many others. However, RAM machines do capture actual computers to a first approximation and so (as we will see), the running time of an algorithm on a RAM machine (e.g., $O(n)$ vs $O(n^2)$) is strongly correlated with its practical efficiency.

8.2 THE GORY DETAILS (OPTIONAL)

We do not show the full formal proof of Theorem 8.1 but focus on the most important parts: implementing indexed access, and simulating two dimensional arrays with one dimensional ones. Even these are already quite tedious to describe, as will not be surprising to anyone that has ever written a compiler. Hence you can feel free to merely skim this section. The important point is not for you to know all details by heart but to be convinced that in principle it *is* possible to transform a NAND-RAM program to an equivalent NAND-TM program, and even be convinced that, with sufficient time and effort, *you* could do it if you wanted to.

8.2.1 Indexed access in NAND-TM

In NAND-TM we can only access our arrays in the position of the index variable `i`, while NAND-RAM has integer-valued variables and can use them for *indexed access* to arrays, of the form `Foo[bar]`. To implement indexed access in NAND-TM, we will encode integers in our arrays using some prefix-free representation (see Section 2.5.2)), and then have a procedure `Setindex(Bar)` that sets `i` to the value encoded by `Bar`. We can simulate the effect of `Foo[Bar]` using `Setindex(Bar)` followed by `Foo[i]`.

Implementing `Setindex(Bar)` can be achieved as follows:

1. We initialize an array `Atzero` such that `Atzero`$[0]=1$ and `Atzero`$[j]=0$ for all $j>0$. (This can be easily done in NAND-TM as all uninitialized variables default to zero.)
2. Set `i` to zero, by decrementing it until we reach the point where `Atzero[i]`$=1$.
3. Let `Temp` be an array encoding the number 0.
4. We use `GOTO` to simulate an inner loop of the form: **while** `Temp` $\neq$ `Bar`, increment `Temp`.
5. At the end of the loop, `i` is equal to the value encoded by `Bar`.

In NAND-TM code (using some syntactic sugar), we can implement the above operations as follows:

```
# assume Atzero is an array such that Atzero[0]=1
# and Atzero[j]=0 for all j>0

# set i to 0.
LABEL("zero_idx")
dir0 = zero
dir1 = one
# corresponds to i <- i-1
GOTO("zero_idx",NOT(Atzero[i]))
...
# zero out temp
#(code below assumes a specific prefix-free encoding in
 ↪  which 10 is the "end marker")
Temp[0] = 1
Temp[1] = 0
# set i to Bar, assume we know how to increment, compare
LABEL("increment_temp")
cond = EQUAL(Temp,Bar)
dir0 = one
dir1 = one
# corresponds to i <- i+1
INC(Temp)
GOTO("increment_temp",cond)
# if we reach this point, i is number encoded by Bar
...
# final instruction of program
MODANDJUMP(dir0,dir1)
```

8.2.2 Two dimensional arrays in NAND-TM

To implement two dimensional arrays, we want to embed them in a one dimensional array. The idea is that we come up with a *one to one*

function $embed : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and so embed the location (i, j) of the two dimensional array `Two` in the location $embed(i, j)$ of the array `One`.

Since the set $\mathbb{N} \times \mathbb{N}$ seems "much bigger" than the set $\mathbb{N}$, a priori it might not be clear that such a one to one mapping exists. However, once you think about it more, it is not that hard to construct. For example, you could ask a child to use scissors and glue to transform a 10" by 10" piece of paper into a 1" by 100" strip. This is essentially a one to one map from $[10] \times [10]$ to $[100]$. We can generalize this to obtain a one to one map from $[n] \times [n]$ to $[n^2]$ and more generally a one to one map from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$. Specifically, the following map $embed$ would do (see Fig. 8.5):

$$embed(x, y) = \tfrac{1}{2}(x + y)(x + y + 1) + x \quad .$$

Exercise 8.3 asks you to prove that $embed$ is indeed one to one, as well as computable by a NAND-TM program. (The latter can be done by simply following the grade-school algorithms for multiplication, addition, and division.) This means that we can replace code of the form `Two[Foo][Bar] = something` (i.e., access the two dimensional array `Two` at the integers encoded by the one dimensional arrays `Foo` and `Bar`) by code of the form:

```
Blah = embed(Foo,Bar)
Setindex(Blah)
Two[i] = something
```

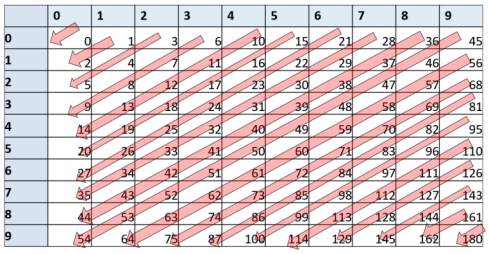


Figure 8.5: Illustration of the map $embed(x, y) = \frac{1}{2}(x + y)(x + y + 1) + x$ for $x, y \in [10]$, one can see that for every distinct pairs (x, y) and (x', y'), $embed(x, y) \neq embed(x', y')$.

8.2.3 All the rest

Once we have two dimensional arrays and indexed access, simulating NAND-RAM with NAND-TM is just a matter of implementing the standard algorithms for arithmetic operations and comparisons in NAND-TM. While this is cumbersome, it is not difficult, and the end result is to show that every NAND-RAM program P can be simulated by an equivalent NAND-TM program Q, thus completing the proof of Theorem 8.1.

R

Remark 8.3 — Recursion in NAND-RAM (advanced). One concept that appears in many programming languages but we did not include in NAND-RAM programs is *recursion*. However, recursion (and function calls in general) can be implemented in NAND-RAM using the stack data structure. A *stack* is a data structure containing a sequence of elements, where we can "push" elements into it and "pop" them from it in "first in last out" order.

We can implement a stack using an array of integers `Stack` and a scalar variable `stackpointer` that will

be the number of items in the stack. We implement `push(foo)` by

```
Stack[stackpointer]=foo
stackpointer += one
```

and implement `bar = pop()` by

```
bar = Stack[stackpointer]
stackpointer -= one
```

We implement a function call to F by pushing the arguments for F into the stack. The code of F will "pop" the arguments from the stack, perform the computation (which might involve making recursive or non-recursive calls) and then "push" its return value into the stack. Because of the "first in last out" nature of a stack, we do not return control to the calling procedure until all the recursive calls are done.

The fact that we can implement recursion using a non-recursive language is not surprising. Indeed, *machine languages* typically do not have recursion (or function calls in general), and hence a compiler implements function calls using a stack and `GOTO`. You can find online tutorials on how recursion is implemented via stack in your favorite programming language, whether it's Python , JavaScript, or Lisp/Scheme.

8.3 TURING EQUIVALENCE (DISCUSSION)

Any of the standard programming languages such as `C`, `Java`, `Python`, `Pascal`, `Fortran` have very similar operations to NAND-RAM. (Indeed, ultimately they can all be executed by machines which have a fixed number of registers and a large memory array.) Hence using Theorem 8.1, we can simulate any program in such a programming language by a NAND-TM program. In the other direction, it is a fairly easy programming exercise to write an interpreter for NAND-TM in any of the above programming languages. Hence we can also simulate NAND-TM programs (and so by Theorem 7.11, Turing machines) using these programming languages. This property of being equivalent in power to Turing machines / NAND-TM is called *Turing Equivalent* (or sometimes *Turing Complete*). Thus all programming languages we are familiar with are Turing equivalent.[1]

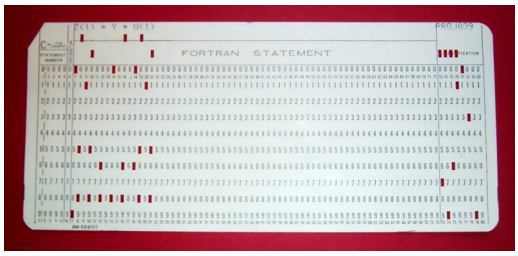


Figure 8.6: A punched card corresponding to a Fortran statement.

8.3.1 The "Best of both worlds" paradigm

The equivalence between Turing machines and RAM machines allows us to choose the most convenient language for the task at hand:

- When we want to *prove a theorem* about all programs/algorithms, we can use Turing machines (or NAND-TM) since they are sim-

[1] Some programming languages have fixed (even if extremely large) bounds on the amount of memory they can access, which formally prevent them from being applicable to computing infinite functions and hence simulating Turing machines. We ignore such issues in this discussion and assume access to some storage device without a fixed upper bound on its capacity.

pler and easier to analyze. In particular, if we want to show that a certain function *cannot* be computed, then we will use Turing machines.

- When we want to show that a function *can be computed* we can use RAM machines or NAND-RAM, because they are easier to program in and correspond more closely to high level programming languages we are used to. In fact, we will often describe NAND-RAM programs in an informal manner, trusting that the reader can fill in the details and translate the high level description to the precise program. (This is just like the way people typically use informal or "pseudocode" descriptions of algorithms, trusting that their audience will know to translate these descriptions to code if needed.)

Our usage of Turing machines / NAND-TM and RAM Machines / NAND-RAM is very similar to the way people use in practice high and low level programming languages. When one wants to produce a device that executes programs, it is convenient to do so for a very simple and "low level" programming language. When one wants to describe an algorithm, it is convenient to use as high level a formalism as possible.

Big Idea 10 Using equivalence results such as those between Turing and RAM machines, we can *"have our cake and eat it too"*. We can use a simpler model such as Turing machines when we want to prove something *can't* be done, and use a feature-rich model such as RAM machines when we want to prove something *can* be done.

Figure 8.7: By having the two equivalent languages NAND-TM and NAND-RAM, we can "have our cake and eat it too", using NAND-TM when we want to prove that programs *can't* do something, and using NAND-RAM or other high level languages when we want to prove that programs *can* do something.

8.3.2 Let's talk about abstractions

"The programmer is in the unique position that ... he has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before.", Edsger Dijkstra, "On the cruelty of really teaching computing science", 1988.

At some point in any theory of computation course, the instructor and students need to have *the talk*. That is, we need to discuss the *level of abstraction* in describing algorithms. In algorithms courses, one typically describes algorithms in English, assuming readers can "fill in the details" and would be able to convert such an algorithm into an implementation if needed. For example, Algorithm 8.4 is a high level description of the breadth first search algorithm.

Algorithm 8.4 — Breadth First Search.

Input: Graph G, vertices u, v

Output: "connected" when u is connected to v in G, "disconnected"

```
Initialize empty queue Q.
Put u in Q
while Q is not empty do
    Remove top vertex w from Q
    if w = v then return "connected"
    end if
    Mark w
    Add all unmarked neighbors of w to Q.
end while
return "disconnected"
```

If we wanted to give more details on how to implement breadth first search in a programming language such as Python or C (or NAND-RAM / NAND-TM for that matter), we would describe how we implement the queue data structure using an array, and similarly how we would use arrays to mark vertices. We call such an "intermediate level" description an *implementation level* or *pseudocode* description. Finally, if we want to describe the implementation precisely, we would give the full code of the program (or another fully precise representation, such as in the form of a list of tuples). We call this a *formal* or *low level* description.

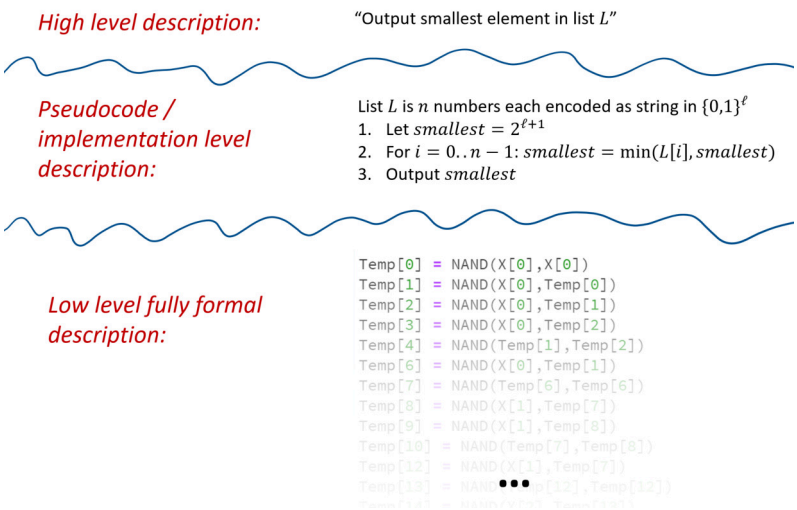


Figure 8.8: We can describe an algorithm at different levels of granularity/detail and precision. At the highest level we just write the idea in words, omitting all details on representation and implementation. In the intermediate level (also known as *implementation* or *pseudocode*) we give enough details of the implementation that would allow someone to derive it, though we still fall short of providing the full code. The lowest level is where the actual code or mathematical description is fully spelled out. These different levels of detail all have their uses, and moving between them is one of the most important skills for a computer scientist.

While we started off by describing NAND-CIRC, NAND-TM, and NAND-RAM programs at the full formal level, as we progress in this

book we will move to implementation and high level description. After all, our goal is not to use these models for actual computation, but rather to analyze the general phenomenon of computation. That said, if you don't understand how the high level description translates to an actual implementation, going "down to the metal" is often an excellent exercise. One of the most important skills for a computer scientist is the ability to move up and down hierarchies of abstractions.

A similar distinction applies to the notion of *representation* of objects as strings. Sometimes, to be precise, we give a *low level specification* of exactly how an object maps into a binary string. For example, we might describe an encoding of n vertex graphs as length n^2 binary strings, by saying that we map a graph G over the vertices $[n]$ to a string $x \in \{0,1\}^{n^2}$ such that the $n \cdot i + j$-th coordinate of x is 1 if and only if the edge $\overrightarrow{i\ j}$ is present in G. We can also use an *intermediate* or *implementation level* description, by simply saying that we represent a graph using the adjacency matrix representation.

Finally, because we are translating between the various representations of graphs (and objects in general) can be done via a NAND-RAM (and hence a NAND-TM) program, when talking in a high level we also suppress discussion of representation altogether. For example, the fact that graph connectivity is a computable function is true regardless of whether we represent graphs as adjacency lists, adjacency matrices, list of edge-pairs, and so on and so forth. Hence, in cases where the precise representation doesn't make a difference, we would often talk about our algorithms as taking as input an object X (that can be a graph, a vector, a program, etc.) without specifying how X is encoded as a string.

Defining "Algorithms". Up until now we have used the term "algorithm" informally. However, Turing machines and the range of equivalent models yield a way to precisely and formally define algorithms. Hence whenever we refer to an *algorithm* in this book, we will mean that it is an instance of one of the Turing equivalent models, such as Turing machines, NAND-TM, RAM machines, etc. Because of the equivalence of all these models, in many contexts, it will not matter which of these we use.

8.3.3 Turing completeness and equivalence, a formal definition (optional)

A *computational model* is some way to define what it means for a *program* (which is represented by a string) to compute a (partial) *function*. A *computational model* $\mathcal{M}$ is *Turing complete* if we can map every Turing machine (or equivalently NAND-TM program) N into a program P for $\mathcal{M}$ that computes the same function as N. It is *Turing equivalent* if the other direction holds as well (i.e., we can map every program in $\mathcal{M}$ to a Turing machine that computes the same function).

We can define this notion formally as follows. (This formal definition is not crucial for the remainder of this book so feel to skip it as long as you understand the general concept of Turing equivalence; This notion is sometimes referred to in the literature as Gödel numbering or admissible numbering.)

Definition 8.5 — Turing completeness and equivalence (optional). Let $\mathcal{F}$ be the set of all partial functions from $\{0,1\}^*$ to $\{0,1\}^*$. A *computational model* is a map $\mathcal{M} : \{0,1\}^* \to \mathcal{F}$.

We say that a program $P \in \{0,1\}^*$ *$\mathcal{M}$-computes* a function $F \in \mathcal{F}$ if $\mathcal{M}(P) = F$.

A computational model $\mathcal{M}$ is *Turing complete* if there is a computable map $ENCODE_{\mathcal{M}} : \{0,1\}^* \to \{0,1\}^*$ for every Turing machine N (represented as a string), $\mathcal{M}(ENCODE_{\mathcal{M}}(N))$ is equal to the partial function computed by N.

A computational model $\mathcal{M}$ is *Turing equivalent* if it is Turing complete and there exists a computable map $DECODE_{\mathcal{M}} : \{0,1\}^* \to \{0,1\}^*$ such that or every string $P \in \{0,1\}^*$, $N = DECODE_{\mathcal{M}}(P)$ is a string representation of a Turing machine that computes the function $\mathcal{M}(P)$.

Some examples of Turing equivalent models (some of which we have already seen, and some are discussed below) include:

- Turing machines
- NAND-TM programs
- NAND-RAM programs
- λ calculus
- Game of life (mapping programs and inputs/outputs to starting and ending configurations)
- Programming languages such as Python/C/Javascript/OCaml... (allowing for unbounded storage)

8.4 CELLULAR AUTOMATA

Many physical systems can be described as consisting of a large number of elementary components that interact with one another. One way to model such systems is using *cellular automata*. This is a system that consists of a large (or even infinite) number of cells. Each cell only has a constant number of possible states. At each time step, a cell updates to a new state by applying some simple rule to the state of itself and its neighbors.

A canonical example of a cellular automaton is Conway's Game of Life. In this automata the cells are arranged in an infinite two dimensional grid. Each cell has only two states: "dead" (which we can

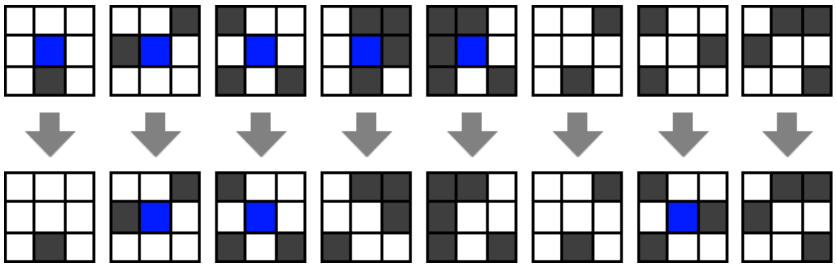


Figure 8.9: Rules for Conway's Game of Life. Image from this blog post.

encode as 0 and identify with $\emptyset$) or "alive" (which we can encode as 1). The next state of a cell depends on its previous state and the states of its 8 vertical, horizontal and diagonal neighbors (see Fig. 8.9). A dead cell becomes alive only if exactly three of its neighbors are alive. A live cell continues to live if it has two or three live neighbors. Even though the number of cells is potentially infinite, we can encode the state using a finite-length string by only keeping track of the live cells. If we initialize the system in a configuration with a finite number of live cells, then the number of live cells will stay finite in all future steps. The Wikipedia page for the Game of Life contains some beautiful figures and animations of configurations that produce very interesting evolutions.

2 dimensional cellular automaton:

⋱	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋰
⋯	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	⋯
⋯	$\emptyset$	a	c	a	b	a	$\emptyset$	⋯
⋯	$\emptyset$	b	a	$\emptyset$	c	a	$\emptyset$	⋯
⋯	$\emptyset$	a	b	c	a	b	$\emptyset$	⋯
⋯	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	⋯
⋰	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

1 dimensional cellular automaton:

⋯	$\emptyset$	a	c	a	b	a	$\emptyset$	⋯

Figure 8.10: In a *two dimensional cellular automaton* every cell is in position i, j for some integers $i, j \in \mathbb{Z}$. The *state* of a cell is some value $A_{i,j} \in \Sigma$ for some finite alphabet Σ. At a given time step, the state of the cell is adjusted according to some function applied to the state of (i, j) and all its neighbors $(i \pm 1, j \pm 1)$. In a *one dimensional cellular automaton* every cell is in position $i \in \mathbb{Z}$ and the state A_i of i at the next time step depends on its current state and the state of its two neighbors $i - 1$ and $i + 1$.

Since the cells in the game of life are are arranged in an infinite two-dimensional grid, it is an example of a *two dimensional cellular automaton*. We can also consider the even simpler setting of a *one dimensional cellular automaton*, where the cells are arranged in an infinite line, see Fig. 8.10. It turns out that even this simple model is enough to achieve

Turing-completeness. We will now formally define one-dimensional cellular automata and then prove their Turing completeness.

Definition 8.6 — One dimensional cellular automata. Let Σ be a finite set containing the symbol $\varnothing$. A *one dimensional cellular automaton* over alphabet Σ is described by a *transition rule* $r : \Sigma^3 \to \Sigma$, which satisfies $r(\varnothing, \varnothing, \varnothing) = \varnothing$.

A *configuration* of the automaton r is a function $A : \mathbb{Z} \to \Sigma$. If an automaton with rule r is in configuration A, then its next configuration, denoted by $A' = NEXT_r(A)$, is the function A' such that $A'(i) = r(A(i-1), A(i), A(i+1))$ for every $i \in \mathbb{Z}$. In other words, the next state of the automaton r at point i is obtained by applying the rule r to the values of A at i and its two neighbors.

Finite configuration. We say that a configuration of an automaton r is *finite* if there is only some finite number of indices $i_0, \ldots, i_{j-1}$ in $\mathbb{Z}$ such that $A(i_j) \neq \varnothing$. (That is, for every $i \notin \{i_0, \ldots, i_{j-1}\}$, $A(i) = \varnothing$.) Such a configuration can be represented using a finite string that encodes the indices $i_0, \ldots, i_{n-1}$ and the values $A(i_0), \ldots, A(i_{n-1})$. Since $R(\varnothing, \varnothing, \varnothing) = \varnothing$, if A is a finite configuration then $NEXT_r(A)$ is finite as well. We will only be interested in studying cellular automata that are initialized in finite configurations, and hence remain in a finite configuration throughout their evolution.

8.4.1 One dimensional cellular automata are Turing complete

We can write a program (for example using NAND-RAM) that simulates the evolution of any cellular automaton from an initial finite configuration by simply storing the values of the cells with state not equal to $\varnothing$ and repeatedly applying the rule r. Hence cellular automata can be simulated by Turing machines. What is more surprising that the other direction holds as well. For example, as simple as its rules seem, we can simulate a Turing machine using the game of life (see Fig. 8.11).

In fact, even one dimensional cellular automata can be Turing complete:

Theorem 8.7 — One dimensional automata are Turing complete. For every Turing machine M, there is a one dimensional cellular automaton that can simulate M on every input x.

To make the notion of "simulating a Turing machine" more precise we will need to define *configurations* of Turing machines. We will do so in Section 8.4.2 below, but at a high level a *configuration* of a Turing machine is a string that encodes its full state at a given step in

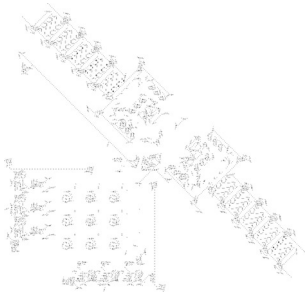


Figure 8.11: A Game-of-Life configuration simulating a Turing machine. Figure by Paul Rendell.

its computation. That is, the contents of all (non-empty) cells of its tape, its current state, as well as the head position.

The key idea in the proof of Theorem 8.7 is that at every point in the computation of a Turing machine M, the only cell in M's tape that can change is the one where the head is located, and the value this cell changes to is a function of its current state and the finite state of M. This observation allows us to encode the configuration of a Turing machine M as a finite configuration of a cellular automaton r, and ensure that a one-step evolution of this encoded configuration under the rules of r corresponds to one step in the execution of the Turing machine M.

8.4.2 Configurations of Turing machines and the next-step function

To turn the above ideas into a rigorous proof (and even statement!) of Theorem 8.7 we will need to precisely define the notion of *configurations* of Turing machines. This notion will be useful for us in later chapters as well.

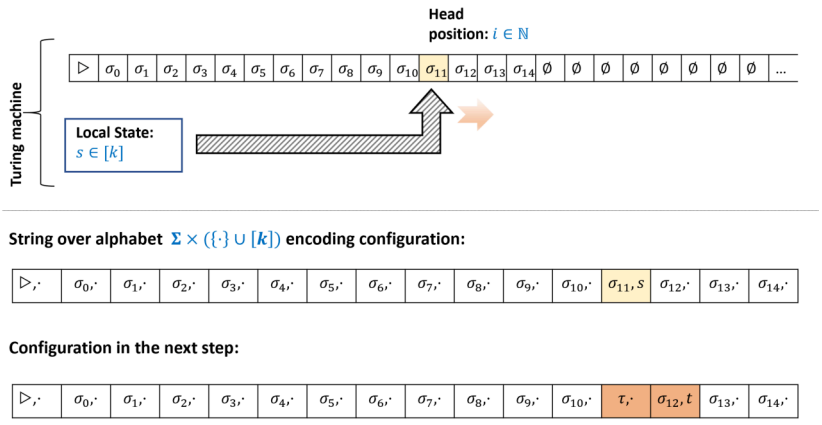


Figure 8.12: A *configuration* of a Turing machine M with alphabet Σ and state space $[k]$ encodes the state of M at a particular step in its execution as a string α over the alphabet $\overline{\Sigma} = \Sigma \times (\{\cdot\} \times [k])$. The string is of length t where t is such that M's tape contains $\varnothing$ in all positions t and larger and M's head is in a position smaller than t. If M's head is in the i-th position, then for $j \neq i$, α_j encodes the value of the j-th cell of M's tape, while α_i encodes both this value as well as the current state of M. If the machine writes the value τ, changes state to t, and moves right, then in the next configuration will contain at position i the value $(\tau, \cdot)$ and at position $i+1$ the value (α_{i+1}, t).

Definition 8.8 — Configuration of Turing Machines.. Let M be a Turing machine with tape alphabet Σ and state space $[k]$. A *configuration of* M is a string $\alpha \in \overline{\Sigma}^*$ where $\overline{\Sigma} = \Sigma \times (\{\cdot\} \cup [k])$ that satisfies that there is exactly one coordinate i for which $\alpha_i = (\sigma, s)$ for some $\sigma \in \Sigma$ and $s \in [k]$. For all other coordinates j, $\alpha_j = (\sigma', \cdot)$ for some $\sigma' \in \Sigma$.

A configuration $\alpha \in \overline{\Sigma}^*$ of M corresponds to the following state of its execution:

- M's tape contains $\alpha_{j,0}$ for all $j < |\alpha|$ and contains $\varnothing$ for all positions that are at least $|\alpha|$, where we let $\alpha_{j,0}$ be the value σ such that $\alpha_j = (\sigma, t)$ with $\sigma \in \Sigma$ and $t \in \{\cdot\} \cup [k]$. (In other words,

since α_j is a pair of an alphabet symbol σ and either a state in $[k]$ or the symbol $\cdot$, $\alpha_{j,0}$ is the first component σ of this pair.)

- M's head is in the unique position i for which α_i has the form (σ, s) for $s \in [k]$, and M's state is equal to s.

Definition 8.8 below has some technical details, but is not actually that deep or complicated. Try to take a moment to stop and think how *you* would encode as a string the state of a Turing machine at a given point in an execution.

Think what are all the components that you need to know in order to be able to continue the execution from this point onwards, and what is a simple way to encode them using a list of finite symbols. In particular, with an eye towards our future applications, try to think of an encoding which will make it as simple as possible to map a configuration at step t to the configuration at step $t+1$.

Definition 8.8 is a little cumbersome, but ultimately a configuration is simply a string that encodes a *snapshot* of the Turing machine at a given point in the execution. (In operating-systems lingo, it is a "core dump".) Such a snapshot needs to encode the following components:

1. The current head position.
2. The full contents of the large scale memory, that is the tape.
3. The contents of the "local registers", that is the state of the machine.

The precise details of how we encode a configuration are not important, but we do want to record the following simple fact:

Lemma 8.9 Let M be a Turing machine and let $NEXT_M : \overline{\Sigma}^* \to \overline{\Sigma}^*$ be the function that maps a configuration of M to the configuration at the next step of the execution. Then for every $i \in \mathbb{N}$, the value of $NEXT_M(\alpha)_i$ only depends on the coordinates $\alpha_{i-1}, \alpha_i, \alpha_{i+1}$.

(For simplicity of notation, above we use the convention that if i is "out of bounds", such as $i < 0$ or $i > |\alpha|$, then we assume that $\alpha_i = (\varnothing, \cdot)$.) We leave proving Lemma 8.9 as Exercise 8.7. The idea behind the proof is simple: if the head is neither in position i nor positions $i-1$ and $i+1$, then the next-step configuration at i will be the same as it was before. Otherwise, we can "read off" the state of the Turing machine and the value of the tape at the head location from the

configuration at i or one of its neighbors and use that to update what the new state at i should be. Completing the full proof is not hard, but doing it is a great way to ensure that you are comfortable with the definition of configurations.

Completing the proof of Theorem 8.7. We can now restate Theorem 8.7 more formally, and complete its proof:

Theorem 8.10 — One dimensional automata are Turing complete (formal statement). For every Turing machine M, if we denote by $\overline{\Sigma}$ the alphabet of its configuration strings, then there is a one-dimensional cellular automaton r over the alphabet $\overline{\Sigma}^*$ such that

$$(NEXT_M(\alpha)) = NEXT_r(\alpha)$$

for every configuration $\alpha \in \overline{\Sigma}^*$ of M (again using the convention that we consider $\alpha_i = \varnothing$ if i is "out of bounds").

Proof. We consider the element $(\varnothing, \cdot)$ of $\overline{\Sigma}$ to correspond to the $\varnothing$ element of the automaton r. In this case, by Lemma 8.9, the function $NEXT_M$ that maps a configuration of M into the next one is in fact a valid rule for a one dimensional automata. ■

The automaton arising from the proof of Theorem 8.10 has a large alphabet, and furthermore one whose size that depends on the machine M that is being simulated. It turns out that one can obtain an automaton with an alphabet of fixed size that is independent of the program being simulated, and in fact the alphabet of the automaton can be the minimal set $\{0,1\}$! See Fig. 8.13 for an example of such an Turing-complete automaton.

R

Remark 8.11 — Configurations of NAND-TM programs. We can use the same approach as Definition 8.8 to define configurations of a *NAND-TM program*. Such a configuration will need to encode:

1. The current value of the variable `i`.
2. For every scalar variable `foo`, the value of `foo`.
3. For every array variable `Bar`, the value `Bar`$[j]$ for every $j \in \{0,\ldots,t-1\}$ where $t-1$ is the largest value that the index variable `i` ever achieved in the computation.

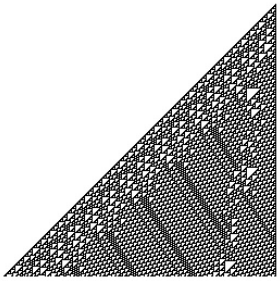


Figure 8.13: Evolution of a one dimensional automata. Each row in the figure corresponds to the configuration. The initial configuration corresponds to the top row and contains only a single "live" cell. This figure corresponds to the "Rule 110" automaton of Stephen Wolfram which is Turing Complete. Figure taken from Wolfram MathWorld.

8.5 LAMBDA CALCULUS AND FUNCTIONAL PROGRAMMING LANGUAGES

The λ calculus is another way to define computable functions. It was proposed by Alonzo Church in the 1930's around the same time as Alan Turing's proposal of the Turing machine. Interestingly, while Turing machines are not used for practical computation, the λ calculus has inspired functional programming languages such as LISP, ML and Haskell, and indirectly the development of many other programming languages as well. In this section we will present the λ calculus and show that its power is equivalent to NAND-TM programs (and hence also to Turing machines). Our Github repository contains a Jupyter notebook with a Python implementation of the λ calculus that you can experiment with to get a better feel for this topic.

The λ operator. At the core of the λ calculus is a way to define "anonymous" functions. For example, instead of giving a name f to a function and defining it as

$$f(x) = x \times x$$

we can write it as

$$\lambda x.x \times x$$

and so $(\lambda x.x \times x)(7) = 49$. That is, you can think of $\lambda x.exp(x)$, where exp is some expression as a way of specifying the anonymous function $x \mapsto exp(x)$. Anonymous functions, using either $\lambda x.f(x)$, $x \mapsto f(x)$ or other closely related notation, appear in many programming languages. For example, in *Python* we can define the squaring function using `lambda x: x*x` while in *JavaScript* we can use `x => x*x` or `(x) => x*x`. In *Scheme* we would define it as `(lambda (x) (* x x))`. Clearly, the name of the argument to a function doesn't matter, and so $\lambda y.y \times y$ is the same as $\lambda x.x \times x$, as both correspond to the squaring function.

Dropping parentheses. To reduce notational clutter, when writing λ calculus expressions we often drop the parentheses for function evaluation. Hence instead of writing $f(x)$ for the result of applying the function f to the input x, we can also write this as simply $f\ x$. Therefore we can write $(\lambda x.x \times x)7 = 49$. In this chapter, we will use both the $f(x)$ and $f\ x$ notations for function application. Function evaluations are associative and bind from left to right, and hence $f\ g\ h$ is the same as $(fg)h$.

8.5.1 Applying functions to functions

A key feature of the λ calculus is that functions are "first-class objects" in the sense that we can use functions as arguments to other functions.

For example, can you guess what number is the following expression equal to?

$$(((\lambda f.(\lambda y.(f\ (f\ y))))(\lambda x.x \times x))\ 3) \tag{8.1}$$

P

The expression (8.1) might seem daunting, but before you look at the solution below, try to break it apart to its components, and evaluate each component at a time. Working out this example would go a long way toward understanding the λ calculus.

Let's evaluate (8.1) one step at a time. As nice as it is for the λ calculus to allow anonymous functions, adding names can be very helpful for understanding complicated expressions. So, let us write $F = \lambda f.(\lambda y.(f(fy)))$ and $g = \lambda x.x \times x$.

Therefore (8.1) becomes

$$((F\ g)\ 3)\ .$$

On input a function f, F outputs the function $\lambda y.(f(f\ \ y))$, or in other words Ff is the function $y \mapsto f(f(y))$. Our function g is simply $g(x) = x^2$ and so (Fg) is the function that maps y to $(y^2)^2 = y^4$. Hence $((Fg)3) = 3^4 = 81$.

Solved Exercise 8.1 What number does the following expression evaluate to?

$$((\lambda x.(\lambda y.x))\ 2)\ 9\ . \tag{8.2}$$

■

Solution:

$\lambda y.x$ is the function that on input y ignores its input and outputs x. Hence $(\lambda x.(\lambda y.x))2$ yields the function $y \mapsto 2$ (or, using λ notation, the function $\lambda y.2$). Hence (8.2) is equivalent to $(\lambda y.2)9 = 2$.

■

8.5.2 Obtaining multi-argument functions via Currying

In a λ expression of the form $\lambda x.e$, the expression e can itself involve the λ operator. Thus for example the function

$$\lambda x.(\lambda y.x + y) \tag{8.3}$$

maps x to the function $y \mapsto x + y$.

In particular, if we invoke the function (8.3) on a to obtain some function f, and then invoke f on b, we obtain the value $a + b$. We

can see that the one-argument function (8.3) corresponding to $a \mapsto (b \mapsto a + b)$ can also be thought of as the two-argument function $(a, b) \mapsto a + b$. Generally, we can use the λ expression $\lambda x.(\lambda y.f(x, y))$ to simulate the effect of a two argument function $(x, y) \mapsto f(x, y)$. This technique is known as Currying. We will use the shorthand $\lambda x, y.e$ for $\lambda x.(\lambda y.e)$. If $f = \lambda x.(\lambda y.e)$ then $(fa)b$ corresponds to applying fa and then invoking the resulting function on b, obtaining the result of replacing in e the occurrences of x with a and occurrences of b with y. By our rules of associativity, this is the same as (fab) which we'll sometimes also write as $f(a, b)$.

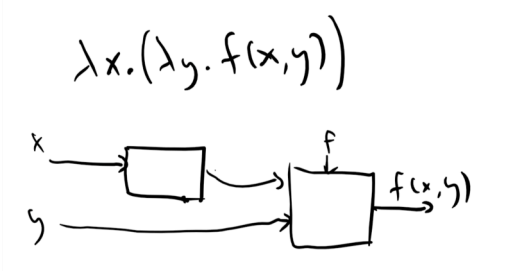


Figure 8.14: In the "currying" transformation, we can create the effect of a two parameter function $f(x, y)$ with the λ expression $\lambda x.(\lambda y.f(x, y))$ which on input x outputs a one-parameter function f_x that has x "hardwired" into it and such that $f_x(y) = f(x, y)$. This can be illustrated by a circuit diagram; see Chelsea Voss's site.

8.5.3 Formal description of the λ calculus

We now provide a formal description of the λ calculus. We start with "basic expressions" that contain a single variable such as x or y and build more complex expressions of the form $(e\ e')$ and $\lambda x.e$ where e, e' are expressions and x is a variable idenifier. Formally λ expressions are defined as follows:

Definition 8.12 — λ expression.. A *λ expression* is either a single variable identifier or an expression e of the one of the following forms:

- **Application:** $e = (e'\ e'')$, where e' and e'' are λ expressions.
- **Abstraction:** $e = \lambda x.(e')$ where e' is a λ expression.

Definition 8.12 is a *recursive definition* since we defined the concept of λ expressions in terms of itself. This might seem confusing at first, but in fact you have known recursive definitions since you were an elementary school student. Consider how we define an *arithmetic expression*: it is an expression that is either just a number, or has one of the forms $(e + e')$, $(e - e')$, $(e \times e')$, or $(e \div e')$, where e and e' are other arithmetic expressions.

Free and bound variables. Variables in a λ expression can either be *free* or *bound* to a λ operator (in the sense of Section 1.4.7). In a single-variable λ expression var, the variable var is free. The set of free and bound variables in an application expression $e = (e'\ e'')$ is the same as that of the underlying expressions e' and e''. In an abstraction expression $e = \lambda var.(e')$, all free occurences of var in e' are bound to the λ operator of e. If you find the notion of free and bound variables confusing, you can avoid all these issues by using unique identifiers for all variables.

Precedence and parentheses. We will use the following rules to allow us to drop some parentheses. Function application associates from left to right, and so fgh is the same as $(fg)h$. Function application has a higher precedence than the λ operator, and so $\lambda x.fgx$ is the same as

$\lambda x.((fg)x)$. This is similar to how we use the precedence rules in arithmetic operations to allow us to use fewer parentheses and so write the expression $(7 \times 3) + 2$ as $7 \times 3 + 2$. As mentioned in Section 8.5.2, we also use the shorthand $\lambda x, y.e$ for $\lambda x.(\lambda y.e)$ and the shorthand $f(x, y)$ for $(f\ x)\ y$. This plays nicely with the "Currying" transformation of simulating multi-input functions using λ expressions.

Equivalence of λ expressions. As we have seen in Solved Exercise 8.1, the rule that $(\lambda x.exp)exp'$ is equivalent to $exp[x \rightarrow exp']$ enables us to modify λ expressions and obtain simpler *equivalent form* for them. Another rule that we can use is that the parameter does not matter and hence for example $\lambda y.y$ is the same as $\lambda z.z$. Together these rules define the notion of *equivalence* of λ expressions:

Definition 8.13 — Equivalence of λ expressions. Two λ expressions are *equivalent* if they can be made into the same expression by repeated applications of the following rules:

1. **Evaluation (aka β reduction):** The expression $(\lambda x.exp)exp'$ is equivalent to $exp[x \rightarrow exp']$.
2. **Variable renaming (aka α conversion):** The expression $\lambda x.exp$ is equivalent to $\lambda y.exp[x \rightarrow y]$.

If exp is a λ expression of the form $\lambda x.exp'$ then it naturally corresponds to the function that maps any input z to $exp'[x \rightarrow z]$. Hence the λ calculus naturally implies a computational model. Since in the λ calculus the inputs can themselves be functions, we need to decide in what order we evaluate an expression such as

$$(\lambda x.f)(\lambda y.gz)\ . \tag{8.4}$$

There are two natural conventions for this:

- *Call by name* (aka *"lazy evaluation"*): We evaluate (8.4) by first plugging in the right-hand expression $(\lambda y.gz)$ as input to the left-hand side function, obtaining $f[x \rightarrow (\lambda y.gz)]$ and then continue from there.
- *Call by value* (aka *"eager evaluation"*): We evaluate (8.4) by first evaluating the right-hand side and obtaining $h = g[y \rightarrow z]$, and then plugging this into the left-hand side to obtain $f[x \rightarrow h]$.

Because the λ calculus has only *pure* functions, that do not have "side effects", in many cases the order does not matter. In fact, it can be shown that if we obtain a definite irreducible expression (for example, a number) in both strategies, then it will be the same one.

However, for concreteness we will always use the "call by name" (i.e., lazy evaluation) order. (The same choice is made in the programming language Haskell, though many other programming languages use eager evaluation.) Formally, the evaluation of a λ expression using "call by name" is captured by the following process:

Definition 8.14 — Simplification of λ expressions. Let e be a λ expression. The *simplification* of e is the result of the following recursive process:

1. If e is a single variable x then the simplification of e is e.
2. If e has the form $e = \lambda x.e'$ then the simplification of e is $\lambda x.f'$ where f' is the simplification of e'.
3. (*Evaluation / β reduction.*) If e has the form $e = (\lambda x.e' \ e'')$ then the simplification of e is the simplification of $e'[x \to e'']$, which denotes replacing all copies of x in e' bound to the λ operator with e''
4. (*Renaming / α conversion.*) The *canonical simplification* of e is obtained by taking the simplification of e and renaming the variables so that the first bound variable in the expression is v_0, the second one is v_1, and so on and so forth.

We say that two λ expressions e and e' are *equivalent*, denoted by $e \cong e'$, if they have the same canonical simplification.

Solved Exercise 8.2 — Equivalence of λ expressions. Prove that the following two expressions e and f are equivalent:

$$e = \lambda x.x$$

$$f = (\lambda a.(\lambda b.b))(\lambda z.zz)$$

Solution:

The canonical simplification of e is simply $\lambda v_0.v_0$. To do the canonical simplification of f we first use β reduction to plug in $\lambda z.zz$ instead of a in $(\lambda b.b)$ but since a is not used in this function at all, we simply obtained $\lambda b.b$ which simplifies to $\lambda v_0.v_0$ as well.

8.5.4 Infinite loops in the λ calculus

Like Turing machines and NAND-TM programs, the simplification process in the λ calculus can also enter into an infinite loop. For example, consider the λ expression

$$\lambda x.xx \ \lambda x.xx \tag{8.5}$$

If we try to simplify (8.5) by invoking the left-hand function on the right-hand one, then we get another copy of (8.5) and hence this never ends. There are examples where the order of evaluation can matter for whether or not an expression can be simplified, see Exercise 8.9.

8.6 THE "ENHANCED" Λ CALCULUS

We now discuss the λ calculus as a computational model. We will start by describing an "enhanced" version of the λ calculus that contains some "superfluous features" but is easier to wrap your head around. We will first show how the enhanced λ calculus is equivalent to Turing machines in computational power. Then we will show how all the features of "enhanced λ calculus" can be implemented as "syntactic sugar" on top of the "pure" (i.e., non-enhanced) λ calculus. Hence the pure λ calculus is equivalent in power to Turing machines (and hence also to RAM machines and all other Turing-equivalent models).

The *enhanced λ calculus* includes the following set of objects and operations:

- **Boolean constants and IF function:** There are λ expressions 0, 1 and *IF* that satisfy the following conditions: for every λ expression e and f, $IF\ 1\ e\ f = e$ and $IF\ 0\ e\ f = f$. That is, *IF* is the function that given three arguments a, e, f outputs e if $a = 1$ and f if $a = 0$.
- **Pairs:** There is a λ expression *PAIR* which we will think of as the *pairing* function. For every λ expressions e, f, $PAIR\ \ e\ \ f$ is the pair $\langle e, f \rangle$ that contains e as its first member and f as its second member. We also have λ expressions *HEAD* and *TAIL* that extract the first and second member of a pair respectively. Hence, for every λ expressions e, f, $HEAD\ (PAIR\ e\ f) = e$ and $TAIL\ (PAIR\ e\ f) = f$.[2]
- **Lists and strings:** There is λ expression *NIL* that corresponds to the *empty list*, which we also denote by $\langle \perp \rangle$. Using *PAIR* and *NIL* we construct *lists*. The idea is that if L is a k element list of the form $\langle e_1, e_2, \ldots, e_k, \perp \rangle$ then for every λ expression e_0 we can obtain the $k+1$ element list $\langle e_0, e_1, e_2, \ldots, e_k, \perp \rangle$ using the expression $PAIR\ \ e_0\ \ L$. For example, for every three λ expressions e, f, g, the following corresponds to the three element list $\langle e, f, g, \perp \rangle$:

$$PAIR\ e\ \ (PAIR\ f\ \ (PAIR\ g\ NIL)) \ .$$

[2] In Lisp, the *PAIR*, *HEAD* and *TAIL* functions are traditionally called `cons`, `car` and `cdr`.

The λ expression *ISEMPTY* returns 1 on *NIL* and returns 0 on every other list. A *string* is simply a list of bits.

- **List operations:** The enhanced λ calculus also contains the *list-processing functions MAP*, *REDUCE*, and *FILTER*. Given a list $L = \langle x_0, \ldots, x_{n-1}, \bot \rangle$ and a function f, *MAP* L f applies f on every member of the list to obtain the new list $L' = \langle f(x_0), \ldots, f(x_{n-1}), \bot \rangle$. Given a list L as above and an expression f whose output is either 0 or 1, *FILTER* L f returns the list $\langle x_i \rangle_{fx_i=1}$ containing all the elements of L for which f outputs 1. The function *REDUCE* applies a "combining" operation to a list. For example, *REDUCE* L $+$ 0 will return the sum of all the elements in the list L. More generally, *REDUCE* takes a list L, an operation f (which we think of as taking two arguments) and a λ expression z (which we think of as the "neutral element" for the operation f, such as 0 for addition and 1 for multiplication). The output is defined via

$$\textit{REDUCE } L \ f \ z = \begin{cases} z & L = \textit{NIL} \\ f\ (\textit{HEAD } L)\ (\textit{REDUCE } (\textit{TAIL } L)\ f\ z) & \text{otherwise} \end{cases} .$$

See Fig. 8.16 for an illustration of the three list-processing operations.

- **Recursion:** Finally, we want to be able to execute *recursive functions*. Since in λ calculus functions are *anonymous*, we can't write a definition of the form $f(x) = blah$ where $blah$ includes calls to f. Instead we use functions f that take an additional input me as a parameter. The operator *RECURSE* will take such a function f as input and return a "recursive version" of f where all the calls to me are replaced by recursive calls to this function. That is, if we have a function F taking two parameters me and x, then *RECURSE* F will be the function f taking one parameter x such that $f(x) = F(f, x)$ for every x.

Solved Exercise 8.3 — Compute NAND using λ calculus. Give a λ expression N such that $N\ x\ y = \textit{NAND}(x, y)$ for every $x, y \in \{0, 1\}$.

■

Solution:

The *NAND* of x, y is equal to 1 unless $x = y = 1$. Hence we can write

$$N = \lambda x, y.\textit{IF}(x, \textit{IF}(y, 0, 1), 1)$$

Solved Exercise 8.4 — Compute XOR using λ calculus. Give a λ expression XOR such that for every list $L = \langle x_0, \ldots, x_{n-1}, \bot \rangle$ where $x_i \in \{0,1\}$ for $i \in [n]$, $XORL$ evaluates to $\sum x_i \mod 2$.

Solution:

First, we note that we can compute XOR of two bits as follows:

$$NOT = \lambda a.IF(a,0,1) \tag{8.6}$$

and

$$XOR_2 = \lambda a,b.IF(b,NOT(a),a) \tag{8.7}$$

(We are using here a bit of syntactic sugar to describe the functions. To obtain the λ expression for XOR we will simply replace the expression (8.6) in (8.7).) Now recursively we can define the XOR of a list as follows:

$$XOR(L) = \begin{cases} 0 & L \text{ is empty} \\ XOR_2(HEAD(L), XOR(TAIL(L))) & \text{otherwise} \end{cases}$$

This means that XOR is equal to

$$RECURSE\ (\lambda me, L.IF(ISEMPTY(L), 0, XOR_2(HEAD\ L\ \ ,\ \ me(TAIL\ L)))) .$$

That is, XOR is obtained by applying the $RECURSE$ operator to the function that on inputs me, L, returns 0 if $ISEMPTY(L)$ and otherwise returns XOR_2 applied to $HEAD(L)$ and to $me(TAIL(L))$.

We could have also computed XOR using the $REDUCE$ operation, we leave working this out as an exercise to the reader.

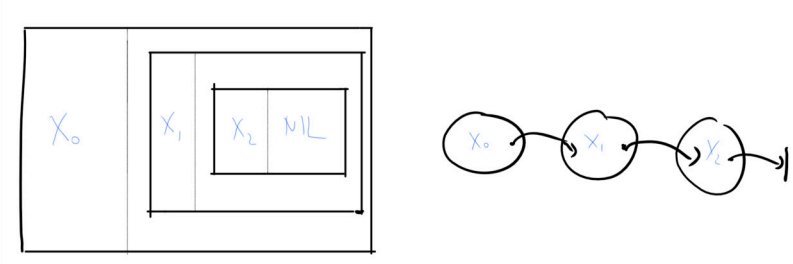


Figure 8.15: A list $\langle x_0, x_1, x_2 \rangle$ in the λ calculus is constructed from the tail up, building the pair $\langle x_2, NIL \rangle$, then the pair $\langle x_1, \langle x_2, NIL \rangle \rangle$ and finally the pair $\langle x_0, \langle x_1, \langle x_2, NIL \rangle \rangle \rangle$. That is, a list is a pair where the first element of the pair is the first element of the list and the second element is the rest of the list. The figure on the left renders this "pairs inside pairs" construction, though it is often easier to think of a list as a "chain", as in the figure on the right, where the second element of each pair is thought of as a *link*, *pointer* or *reference* to the remainder of the list.

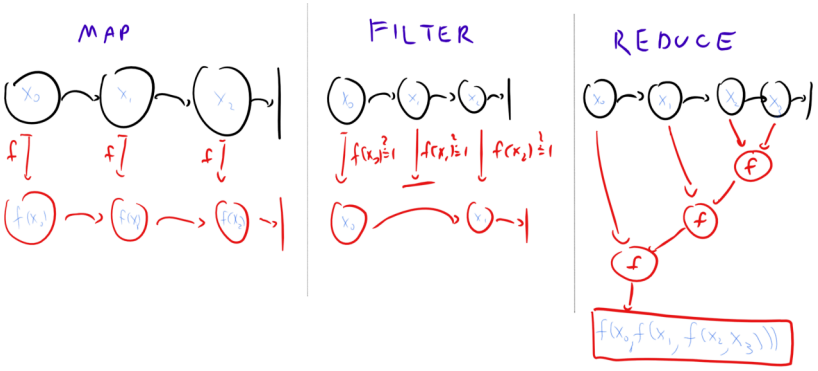


Figure 8.16: Illustration of the *MAP*, *FILTER* and *REDUCE* operations.

8.6.1 Computing a function in the enhanced λ calculus

An *enhanced λ expression* is obtained by composing the objects above with the *application* and *abstraction* rules. The result of simplifying a λ expression is an equivalent expression, and hence if two expressions have the same simplification then they are equivalent.

Definition 8.15 — Computing a function via λ calculus. Let $F : \{0,1\}^* \to \{0,1\}^*$

We say that exp *computes* F if for every $x \in \{0,1\}^*$,

$$exp\langle x_0, \ldots, x_{n-1}, \bot \rangle \cong \langle y_0, \ldots, y_{m-1}, \bot \rangle$$

where $n = |x|$, $y = F(x)$, and $m = |y|$, and the notion of equivalence is defined as per Definition 8.14.

8.6.2 Enhanced λ calculus is Turing-complete

The basic operations of the enhanced λ calculus more or less amount to the Lisp or Scheme programming languages. Given that, it is perhaps not surprising that the enhanced λ-calculus is equivalent to Turing machines:

Theorem 8.16 — Lambda calculus and NAND-TM. For every function $F : \{0,1\}^* \to \{0,1\}^*$, F is computable in the enhanced λ calculus if and only if it is computable by a Turing machine.

Proof Idea:

To prove the theorem, we need to show that **(1)** if F is computable by a λ calculus expression then it is computable by a Turing machine, and **(2)** if F is computable by a Turing machine, then it is computable by an enhanced λ calculus expression.

Showing **(1)** is fairly straightforward. Applying the simplification rules to a λ expression basically amounts to "search and replace"

which we can implement easily in, say, NAND-RAM, or for that matter Python (both of which are equivalent to Turing machines in power). Showing **(2)** essentially amounts to simulating a Turing machine (or writing a NAND-TM interpreter) in a functional programming language such as LISP or Scheme. We give the details below but how this can be done is a good exercise in mastering some functional programming techniques that are useful in their own right.

⋆

Proof of Theorem 8.16. We only sketch the proof. The "if" direction is simple. As mentioned above, evaluating λ expressions basically amounts to "search and replace". It is also a fairly straightforward programming exercise to implement all the above basic operations in an imperative language such as Python or C, and using the same ideas we can do so in NAND-RAM as well, which we can then transform to a NAND-TM program.

For the "only if" direction we need to simulate a Turing machine using a λ expression. We will do so by first showing for every Turing machine M a λ expression to compute the next-step function $NEXT_M : \overline{\Sigma}^* \to \overline{\Sigma}^*$ that maps a configuration of M to the next one (see Section 8.4.2).

A configuration of M is a string $\alpha \in \overline{\Sigma}^*$ for a finite set $\overline{\Sigma}$. We can encode every symbol $\sigma \in \overline{\Sigma}$ by a finite string $\{0,1\}^\ell$, and so we will encode a configuration α in the λ calculus as a list $\langle \alpha_0, \alpha_1, \ldots, \alpha_{m-1}, \bot \rangle$ where α_i is an ℓ-length string (i.e., an ℓ-length list of 0's and 1's) encoding a symbol in $\overline{\Sigma}$.

By Lemma 8.9, for every $\alpha \in \overline{\Sigma}^*$, $NEXT_M(\alpha)_i$ is equal to $r(\alpha_{i-1}, \alpha_i, \alpha_{i+1})$ for some finite function $r : \overline{\Sigma}^3 \to \overline{\Sigma}$. Using our encoding of $\overline{\Sigma}$ as $\{0,1\}^\ell$, we can also think of r as mapping $\{0,1\}^{3\ell}$ to $\{0,1\}^\ell$. By Solved Exercise 8.3, we can compute the *NAND* function, and hence *every* finite function, including r, using the λ calculus. Using this insight, we can compute $NEXT_M$ using the λ calculus as follows. Given a list L encoding the configuration $\alpha_0 \cdots \alpha_{m-1}$, we define the lists L_{prev} and L_{next} encoding the configuration α shifted by one step to the right and left respectively. The next configuration α' is defined as $\alpha'_i = r(L_{prev}[i], L[i], L_{next}[i])$ where we let $L'[i]$ denote the i-th element of L'. This can be computed by recursion (and hence using the enhanced λ calculus' *RECURSE* operator) as follows:

Algorithm 8.17 — $NEXT_M$ using the λ calculus.

Input: List $L = \langle \alpha_0, \alpha_1, \ldots, \alpha_{m-1}, \bot \rangle$ encoding a configuration α.
Output: List L' encoding $NEXT_M(\alpha)$

```
1:  procedure ComputeNext(L_prev, L, L_next)
2:      if then ISEMPTY L_prev
3:          return NIL
4:      end if
5:      a ← HEAD L_prev
6:      if then ISEMPTY L
7:          b ← ∅                          # Encoding of ∅ in {0,1}^ℓ
8:      else
9:          b ← HEAD L
10:     end if
11:     if then ISEMPTY L_next
12:         c ← ∅
13:     else
14:         c ← HEAD L_next
15:     end if
16:     return PAIR r(a,b,c) ComputeNext(TAIL L_prev , TAIL L , TAIL L_next)
17: end procedure
18: L_prev ← PAIR ∅ L          # L_prev = ⟨∅, α_0, ..., α_{m-1}, ⊥⟩
19: L_next ← TAIL L            # L_next = ⟨α_1, ..., α_{m-1}, ⊥}
20: return ComputeNext(L_prev, L, L_next)
```

Once we can compute $NEXT_M$, we can simulate the execution of M on input x using the following recursion. Define $FINAL(\alpha)$ to be the final configuration of M when initialized at configuration α. The function $FINAL$ can be defined recursively as follows:

$$FINAL(\alpha) = \begin{cases} \alpha & \alpha \text{ is halting configuration} \\ FINAL(NEXT_M(\alpha)) & \text{otherwise} \end{cases} .$$

Checking whether a configuration is halting (i.e., whether it is one in which the transition function would output Halt) can be easily implemented in the λ calculus, and hence we can use the *RECURSE* to compute *FINAL*. If we let α^0 be the *initial configuration* of M on input x then we can obtain the output $M(x)$ from $FINAL(\alpha^0)$, hence completing the proof.

■

8.7 FROM ENHANCED TO PURE Λ CALCULUS

While the collection of "basic" functions we allowed for the enhanced λ calculus is smaller than what's provided by most Lisp dialects, coming from NAND-TM it still seems a little "bloated". Can we make do with less? In other words, can we find a subset of these basic operations that can implement the rest?

It turns out that there is in fact a proper subset of the operations of the enhanced λ calculus that can be used to implement the rest. That subset is the empty set. That is, we can implement *all* the operations above using the λ formalism only, even without using 0's and 1's. It's λ's all the way down!

This is a good point to pause and think how you would implement these operations yourself. For example, start by thinking how you could implement *MAP* using *REDUCE*, and then *REDUCE* using *RECURSE* combined with $0, 1, IF, PAIR, HEAD, TAIL, NIL, ISEMPTY$. You can also *PAIR*, *HEAD* and *TAIL* based on $0, 1, IF$. The most challenging part is to implement *RECURSE* using only the operations of the pure λ calculus.

Theorem 8.18 — Enhanced λ calculus equivalent to pure λ calculus.. There are λ expressions that implement the functions 0,1,*IF*,*PAIR*, *HEAD*, *TAIL*, *NIL*, *ISEMPTY*, *MAP*, *REDUCE*, and *RECURSE*.

The idea behind Theorem 8.18 is that we encode 0 and 1 themselves as λ expressions, and build things up from there. This is known as Church encoding, as it was originated by Church in his effort to show that the λ calculus can be a basis for all computation. We will not write the full formal proof of Theorem 8.18 but outline the ideas involved in it:

- We define 0 to be the function that on two inputs x, y outputs y, and 1 to be the function that on two inputs x, y outputs x. We use Currying to achieve the effect of two-input functions and hence $0 = \lambda x.\lambda y.y$ and $1 = \lambda x.\lambda y.x$. (This representation scheme is the common convention for representing `false` and `true` but there are many other alternative representations for 0 and 1 that would have worked just as well.)
- The above implementation makes the *IF* function trivial: $IF(cond, a, b)$ is simply $cond\ a\ b$ since $0ab = b$ and $1ab = a$. We can write $IF = \lambda x.x$ to achieve $IF(cond, a, b) = (((IF cond)a)b) = cond\ a\ b$.

- To encode a pair (x, y) we will produce a function $f_{x,y}$ that has x and y "in its belly" and satisfies $f_{x,y}g = gxy$ for every function g. That is, $PAIR = \lambda x, y.\, (\lambda g.gxy)$. We can extract the first element of a pair p by writing $p1$ and the second element by writing $p0$, and so $HEAD = \lambda p.p1$ and $TAIL = \lambda p.p0$.

- We define *NIL* to be the function that ignores its input and always outputs 1. That is, $NIL = \lambda x.1$. The *ISEMPTY* function checks, given an input p, whether we get 1 if we apply p to the function $zero = \lambda x, y.0$ that ignores both its inputs and always outputs 0. For every valid pair of the form $p = PAIRxy$, $pzero = pxy = 0$ while $NILzero = 1$. Formally, $ISEMPTY = \lambda p.p(\lambda x, y.0)$.

R

Remark 8.19 — Church numerals (optional). There is nothing special about Boolean values. You can use similar tricks to implement *natural numbers* using λ terms. The standard way to do so is to represent the number n by the function $ITER_n$ that on input a function f outputs the function $x \mapsto f(f(\cdots f(x)))$ (n times). That is, we represent the natural number 1 as $\lambda f.f$, the number 2 as $\lambda f.(\lambda x.f(fx))$, the number 3 as $\lambda f.(\lambda x.f(f(fx)))$, and so on and so forth. (Note that this is not the same representation we used for 1 in the Boolean context: this is fine; we already know that the same object can be represented in more than one way.) The number 0 is represented by the function that maps any function f to the identity function $\lambda x.x$. (That is, $0 = \lambda f.(\lambda x.x)$.)

In this representation, we can compute $PLUS(n, m)$ as $\lambda f.\lambda x.(nf)((mf)x)$ and $TIMES(n, m)$ as $\lambda f.n(mf)$. Subtraction and division are trickier, but can be achieved using recursion. (Working this out is a great exercise.)

8.7.1 List processing

Now we come to a bigger hurdle, which is how to implement *MAP*, *FILTER*, *REDUCE* and *RECURSE* in the pure λ calculus. It turns out that we can build *MAP* and *FILTER* from *REDUCE*, and *REDUCE* from *RECURSE*. For example $MAP(L, f)$ is the same as $REDUCE(L, g, NIL)$ where g is the operation that on input x and y, outputs $PAIR(f(x), y)$. (I leave checking this as a (recommended!) exercise for you, the reader.)

We can define $REDUCE(L, f, z)$ recursively, by setting $REDUCE(NIL, f, z) = z$ and stipulating that given a non-empty list L, which we can think of as a pair $(head, rest)$,

$REDUCE(L,f,z) = f(head, REDUCE(rest,f,z)))$. Thus, we might try to write a recursive λ expression for *REDUCE* as follows

$$REDUCE = \lambda L,f,z.IF(ISEMPTY(L),z,fHEAD(L)REDUCE(TAIL(L),f,z)) \; . \tag{8.8}$$

The only fly in this ointment is that the λ calculus does not have the notion of recursion, and so this is an invalid definition. But of course we can use our *RECURSE* operator to solve this problem. We will replace the recursive call to "*REDUCE*" with a call to a function me that is given as an extra argument, and then apply *RECURSE* to this. Thus $REDUCE = RECURSE\ myREDUCE$ where

$$myREDUCE = \lambda me,L,f,z.IF(ISEMPTY(L),z,fHEAD(L)me(TAIL(L),f,z)) \; . \tag{8.9}$$

8.7.2 The Y combinator, or recursion without recursion

Eq. (8.9) means that implementing *MAP*, *FILTER*, and *REDUCE* boils down to implementing the *RECURSE* operator in the pure λ calculus. This is what we do now.

How can we implement recursion without recursion? We will illustrate this using a simple example - the *XOR* function. As shown in Solved Exercise 8.4, we can write the *XOR* function of a list recursively as follows:

$$XOR(L) = \begin{cases} 0 & L \text{ is empty} \\ XOR_2(HEAD(L), XOR(TAIL(L))) & \text{otherwise} \end{cases}$$

where $XOR_2 : \{0,1\}^2 \rightarrow \{0,1\}$ is the XOR on two bits. In *Python* we would write this as

```
def xor2(a,b): return 1-b if a else b
def head(L): return L[0]
def tail(L): return L[1:]

def xor(L): return xor2(head(L),xor(tail(L))) if L else 0

print(xor([0,1,1,0,0,1]))
# 1
```

Now, how could we eliminate this recursive call? The main idea is that since functions can take other functions as input, it is perfectly legal in Python (and the λ calculus of course) to give a function *itself* as input. So, our idea is to try to come up with a *non-recursive* function `tempxor` that takes *two inputs*: a function and a list, and such that `tempxor(tempxor,L)` will output the XOR of `L`!

At this point you might want to stop and try to implement this on your own in Python or any other programming language of your choice (as long as it allows functions as inputs).

Our first attempt might be to simply use the idea of replacing the recursive call by `me`. Let's define this function as `myxor`

```
def myxor(me,L): return xor2(head(L),me(tail(L))) if L
 ↪  else 0
```

Let's test this out:

```
myxor(myxor,[1,0,1])
```

If you do this, you will get the following complaint from the interpreter:
`TypeError: myxor() missing 1 required positional argument`
The problem is that `myxor` expects *two* inputs- a function and a list- while in the call to `me` we only provided a list. To correct this, we modify the call to also provide the function itself:

```
def tempxor(me,L): return xor2(head(L),me(me,tail(L))) if
 ↪  L else 0
```

Note the call `me(me,..)` in the definition of `tempxor`: given a function `me` as input, `tempxor` will actually call the function `me` with itself as the first input. If we test this out now, we see that we actually get the right result!

```
tempxor(tempxor,[1,0,1])
# 0
tempxor(tempxor,[1,0,1,1])
# 1
```

and so we can define `xor(L)` as simply `return tempxor(tempxor,L)`.

The approach above is not specific to XOR. Given a recursive function `f` that takes an input `x`, we can obtain a non-recursive version as follows:

1. Create the function `myf` that takes a pair of inputs `me` and `x`, and replaces recursive calls to `f` with calls to `me`.
2. Create the function `tempf` that converts calls in `myf` of the form `me(x)` to calls of the form `me(me,x)`.

3. The function `f(x)` will be defined as `tempf(tempf,x)`

Here is the way we implement the `RECURSE` operator in Python. It will take a function `myf` as above, and replace it with a function g such that `g(x)=myf(g,x)` for every `x`.

```
def RECURSE(myf):
    def tempf(me,x): return myf(lambda y: me(me,y),x)

    return lambda x: tempf(tempf,x)


xor = RECURSE(myxor)

print(xor([0,1,1,0,0,1]))
# 1

print(xor([1,1,0,0,1,1,1,1]))
# 0
```

From Python to the calculus. In the λ calculus, a two input function g that takes a pair of inputs me, y is written as $\lambda me.(\lambda y.g)$. So the function $y \mapsto me(me, y)$ is simply written as $me\ \ me$ and similarly the function $x \mapsto tempf(tempf, x)$ is simply $tempf\ \ tempf$. (Can you see why?) Therefore the function `tempf` defined above can be written as `λ me. myf(me me)`. This means that if we denote the input of `RECURSE` by f, then $RECURSE\ myf = tempf\ tempf$ where $tempf = \lambda m.f(m\ m)$ or in other words

$$RECURSE = \lambda f.((\lambda m.f(m\ m))\ \ (\lambda m.f(m\ m)))$$

The online appendix contains an implementation of the λ calculus using Python. Here is an implementation of the recursive XOR function from that appendix:[3]

```
# XOR of two bits
XOR2 = λ(a,b)(IF(a,IF(b,_0,_1),b))

# Recursive XOR with recursive calls replaced by m
 ↪  parameter
myXOR = λ(m,l)(IF(ISEMPTY(l),_0,XOR2(HEAD(l),m(TAIL(l)))))

# Recurse operator (aka Y combinator)
RECURSE = λf((λm(f(m*m)))(λm(f(m*m))))

# XOR function
```

[3] Because of specific issues of Python syntax, in this implementation we use `f * g` for applying `f` to `g` rather than `fg`, and use `λx(exp)` rather than `λx.exp` for abstraction. We also use `_0` and `_1` for the λ terms for 0 and 1 so as not to confuse with the Python constants.

```
XOR = RECURSE(myXOR)

#TESTING:

XOR(PAIR(_1,NIL)) # List [1]
# equals 1

XOR(PAIR(_1,PAIR(_0,PAIR(_1,NIL)))) # List [1,0,1]
# equals 0
```

Remark 8.20 — The Y combinator. The *RECURSE* operator above is better known as the Y combinator.

It is one of a family of a *fixed point operators* that given a lambda expression F, find a *fixed point* f of F such that $f = Ff$. If you think about it, *XOR* is the fixed point of $myXOR$ above. *XOR* is the function such that for every x, if plug in *XOR* as the first argument of $myXOR$ then we get back *XOR*, or in other words $XOR = myXOR\ XOR$. Hence finding a *fixed point* for $myXOR$ is the same as applying *RECURSE* to it.

8.8 THE CHURCH-TURING THESIS (DISCUSSION)

> *"[In 1934], Church had been speculating, and finally definitely proposed, that the λ-definable functions are all the effectively calculable functions .... When Church proposed this thesis, I sat down to disprove it ... but, quickly realizing that [my approach failed], I became overnight a supporter of the thesis."*, Stephen Kleene, 1979.

> *"[The thesis is] not so much a definition or to an axiom but ... a natural law."*, Emil Post, 1936.

We have defined functions to be *computable* if they can be computed by a NAND-TM program, and we've seen that the definition would remain the same if we replaced NAND-TM programs by Python programs, Turing machines, λ calculus, cellular automata, and many other computational models. The *Church-Turing thesis* is that this is the only sensible definition of "computable" functions. Unlike the "Physical Extended Church-Turing Thesis" (PECTT) which we saw before, the Church-Turing thesis does not make a concrete physical prediction that can be experimentally tested, but it certainly motivates predictions such as the PECTT. One can think of the Church-Turing Thesis as either advocating a definitional choice, making some prediction about all potential computing devices, or suggesting some laws of nature that constrain the natural world. In Scott Aaronson's

words, "whatever it is, the Church-Turing thesis can only be regarded as extremely successful". No candidate computing device (including quantum computers, and also much less reasonable models such as the hypothetical "closed time curve" computers we mentioned before) has so far mounted a serious challenge to the Church-Turing thesis. These devices might potentially make some computations more *efficient*, but they do not change the difference between what is finitely computable and what is not. (The *extended* Church-Turing thesis, which we discuss in Section 13.3, stipulates that Turing machines capture also the limit of what can be *efficiently* computable. Just like its physical version, quantum computing presents the main challenge to this thesis.)

8.8.1 Different models of computation

We can summarize the models we have seen in the following table:

Table 8.1: Different models for computing finite functions and functions with arbitrary input length.

Computational problems	**Type of model**	**Examples**
Finite functions $f : \{0,1\}^n \rightarrow \{0,1\}^m$	Non-uniform computation (algorithm depends on input length)	Boolean circuits, NAND circuits, straight-line programs (e.g., NAND-CIRC)
Functions with unbounded inputs $F : \{0,1\}^* \rightarrow \{0,1\}^*$	Sequential access to memory	Turing machines, NAND-TM programs
–	Indexed access / RAM	RAM machines, NAND-RAM, modern programming languages
–	Other	Lambda calculus, cellular automata

Later on in Chapter 17 we will study *memory bounded* computation. It turns out that NAND-TM programs with a constant amount of memory are equivalent to the model of *finite automata* (the adjectives "deterministic" or "non-deterministic" are sometimes added as well, this model is also known as *finite state machines*) which in turn captures the notion of *regular languages* (those that can be described by regular expressions), which is a concept we will see in Chapter 10.

Chapter Recap

- While we defined computable functions using Turing machines, we could just as well have done so using many other models, including not just NAND-TM programs but also RAM machines, NAND-RAM, the λ-calculus, cellular automata and many other models.
- Very simple models turn out to be "Turing complete" in the sense that they can simulate arbitrarily complex computation.

8.9 EXERCISES

Exercise 8.1 — Alternative proof for TM/RAM equivalence. Let $SEARCH : \{0,1\}^* \rightarrow \{0,1\}^*$ be the following function. The input is a pair (L,k) where $k \in \{0,1\}^*$, L is an encoding of a list of *key value* pairs $(k_0,v_1),\ldots,(k_{m-1},v_{m-1})$ where $k_0,\ldots,k_{m-1}$, $v_0,\ldots,v_{m-1}$ are binary strings. The output is v_i for the smallest i such that $k_i = k$, if such i exists, and otherwise the empty string.

1. Prove that *SEARCH* is computable by a Turing machine.
2. Let $UPDATE(L,k,v)$ be the function whose input is a list L of pairs, and whose output is the list L' obtained by prepending the pair (k,v) to the beginning of L. Prove that *UPDATE* is computable by a Turing machine.
3. Suppose we encode the configuration of a NAND-RAM program by a list L of key/value pairs where the key is either the name of a scalar variable `foo` or of the form `Bar[<num>]` for some number `<num>` and it contains all the non-zero values of variables. Let $NEXT(L)$ be the function that maps a configuration of a NAND-RAM program at one step to the configuration in the next step. Prove that *NEXT* is computable by a Turing machine (you don't have to implement each one of the arithmetic operations: it is enough to implement addition and multiplication).
4. Prove that for every $F : \{0,1\}^* \rightarrow \{0,1\}^*$ that is computable by a NAND-RAM program, F is computable by a Turing machine.

■

Exercise 8.2 — NAND-TM lookup. This exercise shows part of the proof that NAND-TM can simulate NAND-RAM. Produce the code of a NAND-TM program that computes the function $LOOKUP : \{0,1\}^* \rightarrow \{0,1\}$ that is defined as follows. On input $pf(i)x$, where $pf(i)$ denotes a prefix-free encoding of an integer i, $LOOKUP(pf(i)x) = x_i$ if $i < |x|$

and $LOOKUP(pf(i)x) = 0$ otherwise. (We don't care what *LOOKUP* outputs on inputs that are not of this form.) You can choose any prefix-free encoding of your choice, and also can use your favorite programming language to produce this code.

Exercise 8.3 — Pairing. Let $embed : \mathbb{N}^2 \to \mathbb{N}$ be the function defined as $embed(x_0, x_1) = \frac{1}{2}(x_0 + x_1)(x_0 + x_1 + 1) + x_1$.

1. Prove that for every $x^0, x^1 \in \mathbb{N}$, $embed(x^0, x^1)$ is indeed a natural number.
2. Prove that $embed$ is one-to-one
3. Construct a NAND-TM program P such that for every $x^0, x^1 \in \mathbb{N}$, $P(pf(x^0)pf(x^1)) = pf(embed(x^0, x^1))$, where pf is the prefix-free encoding map defined above. You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter.
4. Construct NAND-TM programs P_0, P_1 such that for every $x^0, x^1 \in \mathbb{N}$ and $i \in N$, $P_i(pf(embed(x^0, x^1))) = pf(x^i)$. You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter.

Exercise 8.4 — Shortest Path. Let $SHORTPATH : \{0,1\}^* \to \{0,1\}^*$ be the function that on input a string encoding a triple (G, u, v) outputs a string encoding ∞ if u and v are disconnected in G or a string encoding the length k of the shortest path from u to v. Prove that *SHORTPATH* is computable by a Turing machine. See footnote for hint.[4]

Exercise 8.5 — Longest Path. Let $LONGPATH : \{0,1\}^* \to \{0,1\}^*$ be the function that on input a string encoding a triple (G, u, v) outputs a string encoding ∞ if u and v are disconnected in G or a string encoding the length k of the *longest simple path* from u to v. Prove that *LONGPATH* is computable by a Turing machine. See footnote for hint.[5]

Exercise 8.6 — Shortest path λ expression. Let *SHORTPATH* be as in Exercise 8.4. Prove that there exists a λ expression that computes *SHORTPATH*. You can use Exercise 8.4

Exercise 8.7 — Next-step function is local. Prove Lemma 8.9 and use it to complete the proof of Theorem 8.7.

[4] You don't have to give a full description of a Turing machine: use our "have the cake and eat it too" paradigm to show the existence of such a machine by arguing from more powerful equivalent models.

[5] Same hint as Exercise 8.5 applies. Note that for showing that *LONGPATH* is computable you don't have to give an *efficient* algorithm.

Exercise 8.8 — λ calculus requires at most three variables. Prove that for every λ-expression e with no free variables there is an equivalent λ-expression f that only uses the variables x,y, and z.[6]

Exercise 8.9 — Evaluation order example in λ calculus.

1. Let $e = \lambda x.7\,((\lambda x.xx)(\lambda x.xx))$. Prove that the simplification process of e ends in a definite number if we use the "call by name" evaluation order while it never ends if we use the "call by value" order.

2. (bonus, challenging) Let e be any λ expression. Prove that if the simplification process ends in a definite number if we use the "call by value" order then it also ends in such a number if we use the "call by name" order. See footnote for hint.[7]

Exercise 8.10 — Zip function. Give an enhanced λ calculus expression to compute the function zip that on input a pair of lists I and L of the same length n, outputs a *list of* n *pairs* M such that the j-th element of M (which we denote by M_j) is the pair (I_j, L_j). Thus zip "zips together" these two lists of elements into a single list of pairs.[8]

Exercise 8.11 — Next-step function without $RECURSE$. Let M be a Turing machine. Give an enhanced λ calculus expression to compute the next-step function $NEXT_M$ of M (as in the proof of Theorem 8.16) *without using RECURSE*. See footnote for hint.[9]

Exercise 8.12 — λ calculus to NAND-TM compiler (challenging). Give a program in the programming language of your choice that takes as input a λ expression e and outputs a NAND-TM program P that computes the same function as e. For partial credit you can use the `GOTO` and all NAND-CIRC syntactic sugar in your output program. You can use any encoding of λ expressions as binary string that is convenient for you. See footnote for hint.[10]

Exercise 8.13 — At least two in λ calculus. Let $1 = \lambda x, y.x$ and $0 = \lambda x, y.y$ as before. Define

$$ALT = \lambda a, b, c.(a(b1(c10))(bc0))$$

Prove that ALT is a λ expression that computes the *at least two* function. That is, for every $a, b, c \in \{0, 1\}$ (as encoded above) $ALTabc = 1$ if and only at least two of $\{a, b, c\}$ are equal to 1.

[6] **Hint:** You can reduce the number of variables a function takes by "pairing them up". That is, define a λ expression *PAIR* such that for every x, y $PAIRxy$ is some function f such that $f0 = x$ and $f1 = y$. Then use *PAIR* to iteratively reduce the number of variables used.

[7] Use structural induction on the expression e.

[8] The name zip is a common name for this operation, for example in Python. It should not be confused with the `zip` compression file format.

[9] Use *MAP* and *REDUCE* (and potentially *FILTER*). You might also find the function zip of Exercise 8.10 useful.

[10] Try to set up a procedure such that if array `Left` contains an encoding of a λ expression $\lambda x.e$ and array `Right` contains an encoding of another λ expression e', then the array `Result` will contain $e[x \to e']$.

Exercise 8.14 — Locality of next-step function. This question will help you get a better sense of the notion of *locality of the next step function* of Turing machines. This locality plays an important role in results such as the Turing completeness of λ calculus and one dimensional cellular automata, as well as results such as Godel's Incompleteness Theorem and the Cook Levin theorem that we will see later in this course. Define `STRINGS` to be the a programming language that has the following semantics:

- A `STRINGS` program Q has a single string variable `str` that is both the input and the output of Q. The program has no loops and no other variables, but rather consists of a sequence of conditional search and replace operations that modify `str`.
- The operations of a `STRINGS` program are:
  - `REPLACE(pattern1,pattern2)` where `pattern1` and `pattern2` are fixed strings. This replaces the first occurrence of `pattern1` in `str` with `pattern2`
  - `if search(pattern) { code }` executes `code` if `pattern` is a substring of `str`. The code `code` can itself include nested `if`'s. (One can also add an `else { ... }` to execute if `pattern` is *not* a substring of condf).
  - the returned value is `str`
- A `STRING` program Q computes a function $F : \{0,1\}^* \to \{0,1\}^*$ if for every $x \in \{0,1\}^*$, if we initialize `str` to x and then execute the sequence of instructions in Q, then at the end of the execution `str` equals $F(x)$.

For example, the following is a `STRINGS` program that computes the function $F : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, if x contains a substring of the form $y = 11ab11$ where $a, b \in \{0,1\}$, then $F(x) = x'$ where x' is obtained by replacing the first occurrence of y in x with 00.

```
if search('110011') {
    replace('110011','00')
} else if search('110111') {
    replace('110111','00')
} else if search('111011') {
    replace('111011','00')
} else if search('111111') {
    replace('1111111','00')
}
```

Prove that for every Turing machine program M, there exists a `STRINGS` program Q that computes the $NEXT_M$ function that maps every string encoding a valid *configuration* of M to the string encoding the configuration of the next step of M's computation. (We don't care what the function will do on strings that do not encode a valid configuration.) You don't have to write the `STRINGS` program fully, but you do need to give a convincing argument that such a program exists.

■

8.10 BIBLIOGRAPHICAL NOTES

Chapters 7 in the wonderful book of Moore and Mertens [MM11] contains a great exposition much of this material. .

The RAM model can be very useful in studying the concrete complexity of practical algorithms. Its theoretical study was initiated in [CR73]. However, the exact set of operations that are allowed in the RAM model and their costs vary between texts and contexts. One needs to be careful in making such definitions, especially if the word size grows, as was already shown by Shamir [Sha79]. Chapter 3 in Savage's book [Sav98] contains a more formal description of RAM machines, see also the paper [Hag98]. A study of RAM algorithms that are independent of the input size (known as the "transdichotomous RAM model") was initiated by [FW93]

The models of computation we considered so far are inherently sequential, but these days much computation happens in parallel, whether using multi-core processors or in massively parallel distributed computation in data centers or over the Internet. Parallel computing is important in practice, but it does not really make much difference for the question of what can and can't be computed. After all, if a computation can be performed using m machines in t time, then it can be computed by a single machine in time mt.

The λ-calculus was described by Church in [Chu41]. Pierce's book [Pie02] is a canonical textbook, see also [Bar84]. The "Currying technique" is named after the logician Haskell Curry (the *Haskell* programming language is named after Haskell Curry as well). Curry himself attributed this concept to Moses Schönfinkel, though for some reason the term "Schönfinkeling" never caught on.

Unlike most programming languages, the pure λ-calculus doesn't have the notion of *types*. Every object in the λ calculus can also be thought of as a λ expression and hence as a function that takes one input and returns one output. All functions take one input and return one output, and if you feed a function an input of a form it didn't expect, it still evaluates the λ expression via "search and replace", replacing all instances of its parameter with copies of the input expres-

sion you fed it. Typed variants of the λ calculus are objects of intense research, and are strongly related to type systems for programming language and computer-verifiable proof systems, see [Pie02]. Some of the typed variants of the λ calculus do not have infinite loops, which makes them very useful as ways of enabling static analysis of programs as well as computer-verifiable proofs. We will come back to this point in Chapter 10 and Chapter 22.

Tao has proposed showing the Turing completeness of fluid dynamics (a "water computer") as a way of settling the question of the behavior of the Navier-Stokes equations, see this popular article.

Learning Objectives:

- **The universal machine/program - "one program to rule them all"**
- **A fundamental result in computer science and mathematics: the existence of uncomputable functions.**
- **The *halting problem*: the canonical example of an uncomputable function.**
- **Introduction to the technique of *reductions*.**
- **Rice's Theorem: A "meta tool" for uncomputability results, and a starting point for much of the research on compilers, programming languages, and software verification.**

9

Universality and uncomputability

> *"A function of a variable quantity is an analytic expression composed in any way whatsoever of the variable quantity and numbers or constant quantities."*, Leonhard Euler, 1748.

> *"The importance of the universal machine is clear. We do not need to have an infinity of different machines doing different jobs. ... The engineering problem of producing various machines for various jobs is replaced by the office work of 'programming' the universal machine"*, Alan Turing, 1948

One of the most significant results we showed for Boolean circuits (or equivalently, straight-line programs) is the notion of *universality*: there is a single circuit that can evaluate all other circuits. However, this result came with a significant caveat. To evaluate a circuit of s gates, the universal circuit needed to use a number of gates *larger* than s. It turns out that uniform models such as Turing machines or NAND-TM programs allow us to "break out of this cycle" and obtain a truly *universal Turing machine* U that can evaluate all other machines, including machines that are more complex (e.g., more states) than U itself. (Similarly, there is a *Universal NAND-TM program* U' that can evaluate all NAND-TM programs, including programs that have more lines than U'.)

It is no exaggeration to say that the existence of such a universal program/machine underlies the information technology revolution that began in the latter half of the 20th century (and is still ongoing). Up to that point in history, people have produced various special-purpose calculating devices such as the abacus, the slide ruler, and machines that compute various trigonometric series. But as Turing (who was perhaps the one to see most clearly the ramifications of universality) observed, a *general purpose computer* is much more powerful. Once we build a device that can compute the single universal function, we have the ability, *via software*, to extend it to do arbitrary computations. For example, if we want to simulate a new Turing machine M, we do not need to build a new physical machine, but rather

can represent M as a string (i.e., using *code*) and then input M to the universal machine U.

Beyond the practical applications, the existence of a universal algorithm also has surprising theoretical ramifications, and in particular can be used to show the existence of *uncomputable functions*, upending the intuitions of mathematicians over the centuries from Euler to Hilbert. In this chapter we will prove the existence of the universal program, and also show its implications for uncomputability, see Fig. 9.1

This chapter: A non-mathy overview

In this chapter we will see two of the most important results in Computer Science:

1. The existence of a *universal Turing machine*: a single algorithm that can evaluate all other algorithms,
2. The existence of *uncomputable functions*: functions (including the famous "Halting problem") that *cannot be computed* by any algorithm.

Along the way, we develop the technique of *reductions* as a way to show hardness of computing a function. A *reduction* gives a way to compute a certain function using "wishful thinking" and assuming that another function can be computed. Reductions are of course widely used in programming - we often obtain an algorithm for one task by using another task as a "black box" subroutine. However we will use it in the "contra positive": rather than using a reduction to show that the former task is "easy", we use them to show that the latter task is "hard". Don't worry if you find this confusing - reductions *are* initially confusing - but they can be mastered with time and practice.

9.1 UNIVERSALITY OR A META-CIRCULAR EVALUATOR

We start by proving the existence of a *universal Turing machine*. This is a single Turing machine U that can evaluate *arbitrary* Turing machines M on *arbitrary* inputs x, including machines M that can have more states and larger alphabet than U itself. In particular, U can even be used to evaluate itself! This notion of *self reference* will appear time and again in this book, and as we will see, leads to several counter-intuitive phenomena in computing.

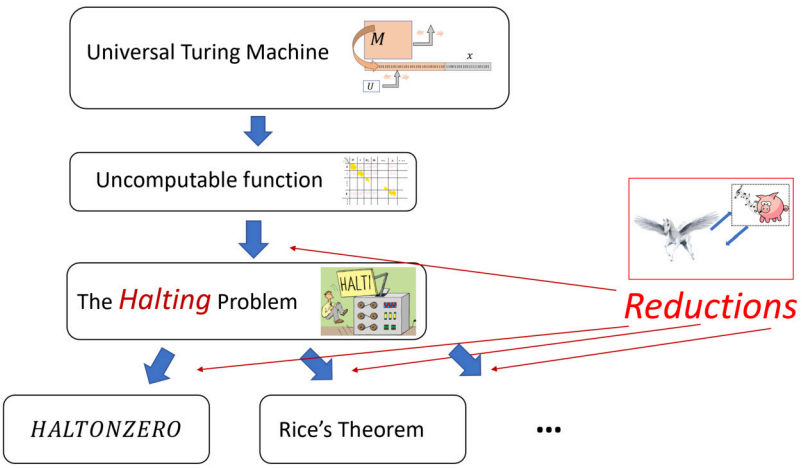


Figure 9.1: In this chapter we will show the existence of a *universal Turing machine* and then use this to derive first the existence of *some* uncomputable function. We then use this to derive the uncomputability of Turing's famous "halting problem" (i.e., the *HALT* function), from which a host of other uncomputability results follow. We also introduce *reductions*, which allow us to use the uncomputability of a function F to derive the uncomputability of a new function G.

Theorem 9.1 — Universal Turing Machine. There exists a Turing machine U such that on every string M which represents a Turing machine, and $x \in \{0,1\}^*$, $U(M,x) = M(x)$.

That is, if the machine M halts on x and outputs some $y \in \{0,1\}^*$ then $U(M,x) = y$, and if M does not halt on x (i.e., $M(x) = \perp$) then $U(M,x) = \perp$.

Big Idea 11 There is a *"universal"* algorithm that can evaluate arbitrary algorithms on arbitrary inputs.

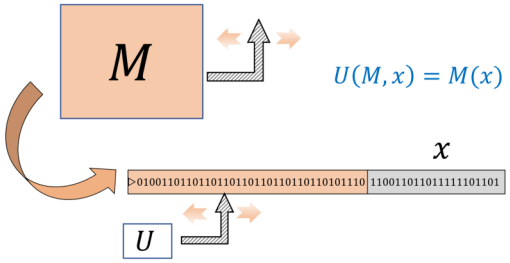


Figure 9.2: A *Universal Turing Machine* is a single Turing Machine U that can evaluate, given input the (description as a string of) arbitrary Turing machine M and input x, the output of M on x. In contrast to the universal circuit depicted in Fig. 5.6, the machine M can be much more complex (e.g., more states or tape alphabet symbols) than U.

Proof Idea:

Once you understand what the theorem says, it is not that hard to prove. The desired program U is an *interpreter* for Turing machines. That is, U gets a representation of the machine M (think of it as source code), and some input x, and needs to simulate the execution of M on x.

Think of how you would code U in your favorite programming language. First, you would need to decide on some representation scheme for M. For example, you can use an array or a dictionary to encode M's transition function. Then you would use some data structure, such as a list, to store the contents of M's tape. Now you can simulate M step by step, updating the data structure as you go along. The interpreter will continue the simulation until the machine halts.

Once you do that, translating this interpreter from your favorite programming language to a Turing machine can be done just as we have seen in Chapter 8. The end result is what's known as a "meta-

circular evaluator": an interpreter for a programming language in the same one. This is a concept that has a long history in computer science starting from the original universal Turing machine. See also Fig. 9.3.

⋆

9.1.1 Proving the existence of a universal Turing Machine

To prove (and even properly state) Theorem 9.1, we need to fix some representation for Turing machines as strings. One potential choice for such a representation is to use the equivalence between Turing machines and NAND-TM programs and hence represent a Turing machine M using the ASCII encoding of the source code of the corresponding NAND-TM program P. However, we will use a more direct encoding.

Definition 9.2 — String representation of Turing Machine. Let M be a Turing machine with k states and a size ℓ alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_{\ell-1}\}$ (we use the convention $\sigma_0 = 0$, $\sigma_1 = 1$, $\sigma_2 = \varnothing$, $\sigma_3 = \triangleright$). We represent M as the triple (k, ℓ, T) where T is the table of values for δ_M:

$$T = (\delta_M(0,\sigma_0), \delta_M(0,\sigma_1), \ldots, \delta_M(k-1,\sigma_{\ell-1})) \ ,$$

where each value $\delta_M(s,\sigma)$ is a triple (s',σ',d) with $s' \in [k]$, $\sigma' \in \Sigma$ and d a number $\{0,1,2,3\}$ encoding one of $\{\mathsf{L},\mathsf{R},\mathsf{S},\mathsf{H}\}$. Thus such a machine M is encoded by a list of $2 + 3k \cdot \ell$ natural numbers. The *string representation* of M is obtained by concatenating a prefix free representation of all these integers. If a string $\alpha \in \{0,1\}^*$ does not represent a list of integers in the form above, then we treat it as representing the trivial Turing machine with one state that immediately halts on every input.

R

Remark 9.3 — Take away points of representation. The details of the representation scheme of Turing machines as strings are immaterial for almost all applications. What you need to remember are the following points:

1. We can represent every Turing machine as a string.
2. Given the string representation of a Turing machine M and an input x, we can simulate M's execution on the input x. (This is the content of Theorem 9.1.)

An additional minor issue is that for convenience we make the assumption that *every* string represents *some* Turing machine. This is very easy to ensure by just mapping strings that would otherwise not represent a

Turing machine into some fixed trivial machine. This assumption is not very important, but does make a few results (such as Rice's Theorem: Theorem 9.15) a little less cumbersome to state.

Using this representation, we can formally prove Theorem 9.1.

Proof of Theorem 9.1. We will only sketch the proof, giving the major ideas. First, we observe that we can easily write a *Python* program that, on input a representation (k, ℓ, T) of a Turing machine M and an input x, evaluates M on X. Here is the code of this program for concreteness, though you can feel free to skip it if you are not familiar with (or interested in) Python:

```
# constants
def EVAL(δ,x):
    '''Evaluate TM given by transition table δ
    on input x'''
    Tape = [" "] + [a for a in x]
    i = 0; s = 0 # i = head pos, s = state
    while True:
        s, Tape[i], d = δ[(s,Tape[i])]
        if d == "H": break
        if d == "L": i = max(i-1,0)
        if d == "R": i += 1
        if i>= len(Tape): Tape.append('Φ')

    j = 1; Y = [] # produce output
    while Tape[j] != 'Φ':
        Y.append(Tape[j])
        j += 1
    return Y
```

On input a transition table δ this program will simulate the corresponding machine M step by step, at each point maintaining the invariant that the array `Tape` contains the contents of M's tape, and the variable `s` contains M's current state.

The above does not prove the theorem as stated, since we need to show a *Turing machine* that computes *EVAL* rather than a Python program. With enough effort, we can translate this Python code line by line to a Turing machine. However, to prove the theorem we don't need to do this, but can use our "eat the cake and have it too" paradigm. That is, while we need to evaluate a Turing machine, in writing the code for the interpreter we are allowed to use a richer model such as NAND-RAM since it is equivalent in power to Turing machines per Theorem 8.1.

Translating the above Python code to NAND-RAM is truly straightforward. The only issue is that NAND-RAM doesn't have the *dictionary* data structure built in, which we have used above to store the transition function δ. However, we can represent a dictionary D of the form $\{key_0 : val_0, \ldots, key_{m-1} : val_{m-1}\}$ as simply a list of pairs. To compute $D[k]$ we can scan over all the pairs until we find one of the form (k, v) in which case we return v. Similarly we scan the list to update the dictionary with a new value, either modifying it or appending the pair (key, val) at the end.

■

R

Remark 9.4 — Efficiency of the simulation. The argument in the proof of Theorem 9.1 is a very inefficient way to implement the dictionary data structure in practice, but it suffices for the purpose of proving the theorem. Reading and writing to a dictionary of m values in this implementation takes $\Omega(m)$ steps, but it is in fact possible to do this in $O(\log m)$ steps using a *search tree* data structure or even $O(1)$ (for "typical" instances) using a *hash table*. NAND-RAM and RAM machines correspond to the architecture of modern electronic computers, and so we can implement hash tables and search trees in NAND-RAM just as they are implemented in other programming languages.

The construction above yields a universal Turing machine with a very large number of states. However, since universal Turing machines have such a philosophical and technical importance, researchers have attempted to find the smallest possible universal Turing machines, see Section 9.7.

9.1.2 Implications of universality (discussion)

There is more than one Turing machine U that satisfies the conditions of Theorem 9.1, but the existence of even a single such machine is already extremely fundamental to both the theory and practice of computer science. Theorem 9.1's impact reaches beyond the particular model of Turing machines. Because we can simulate every Turing machine by a NAND-TM program and vice versa, Theorem 9.1 immediately implies there exists a universal NAND-TM program P_U such that $P_U(P, x) = P(x)$ for every NAND-TM program P. We can also "mix and match" models. For example since we can simulate every NAND-RAM program by a Turing machine, and every Turing machine by the λ calculus, Theorem 9.1 implies that there exists a λ expression e such that for every NAND-RAM program P and input x on which $P(x) = y$, if we encode (P, x) as a λ-expression f (using the

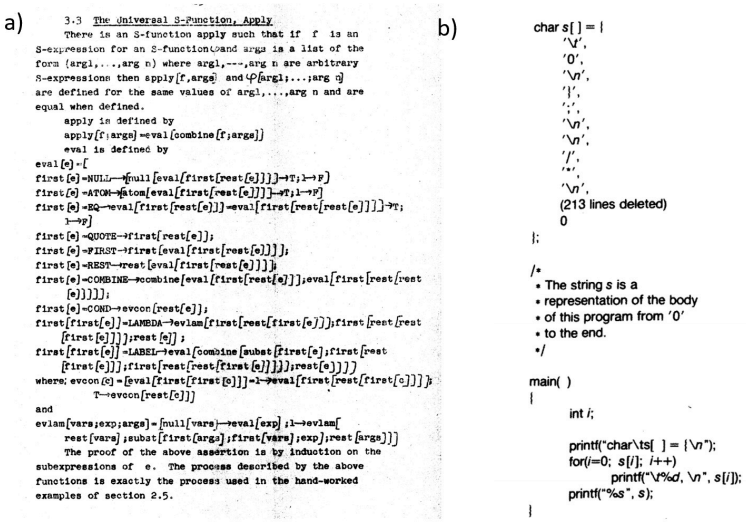


Figure 9.3: **a**) A particularly elegant example of a "meta-circular evaluator" comes from John McCarthy's 1960 paper, where he defined the Lisp programming language and gave a Lisp function that evaluates an arbitrary Lisp program (see above). Lisp was not initially intended as a practical programming language and this example was merely meant as an illustration that the Lisp universal function is more elegant than the universal Turing machine. It was McCarthy's graduate student Steve Russell who suggested that it can be implemented. As McCarthy later recalled, *"I said to him, ho, ho, you're confusing theory with practice, this eval is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the eval in my paper into IBM 704 machine code, fixing a bug, and then advertised this as a Lisp interpreter, which it certainly was"*. **b**) A self-replicating C program from the classic essay of Thompson [Tho84].

λ-calculus encoding of strings as lists of 0's and 1's) then $(e\ f)$ evaluates to an encoding of y. More generally we can say that for every $\mathcal{X}$ and $\mathcal{Y}$ in the set { Turing machines, RAM Machines, NAND-TM, NAND-RAM, λ-calculus, JavaScript, Python, ... } of Turing equivalent models, there exists a program/machine in $\mathcal{X}$ that computes the map $(P, x) \mapsto P(x)$ for every program/machine $P \in \mathcal{Y}$.

The idea of a "universal program" is of course not limited to theory. For example compilers for programming languages are often used to compile *themselves*, as well as programs more complicated than the compiler. (An extreme example of this is Fabrice Bellard's Obfuscated Tiny C Compiler which is a C program of 2048 bytes that can compile a large subset of the C programming language, and in particular can compile itself.) This is also related to the fact that it is possible to write a program that can print its own source code, see Fig. 9.3. There are universal Turing machines known that require a very small number of states or alphabet symbols, and in particular there is a universal Turing machine (with respect to a particular choice of representing Turing machines as strings) whose tape alphabet is $\{\triangleright, \varnothing, 0, 1\}$ and has fewer than 25 states (see Section 9.7).

9.2 IS EVERY FUNCTION COMPUTABLE?

In Theorem 4.12, we saw that NAND-CIRC programs can compute every finite function $f : \{0,1\}^n \to \{0,1\}$. Therefore a natural guess is that NAND-TM programs (or equivalently, Turing machines) could compute every infinite function $F : \{0,1\}^* \to \{0,1\}$. However, this turns out to be *false*. That is, there exists a function $F : \{0,1\}^* \to \{0,1\}$ that is *uncomputable*!

The existence of uncomputable functions is quite surprising. Our intuitive notion of a "function" (and the notion most mathematicians had until the 20th century) is that a function f defines some implicit or explicit way of computing the output $f(x)$ from the input x. The notion of an "uncomputable function" thus seems to be a contradiction in terms, but yet the following theorem shows that such creatures do exist:

Theorem 9.5 — Uncomputable functions. There exists a function $F^* : \{0,1\}^* \to \{0,1\}$ that is not computable by any Turing machine.

Proof Idea:

The idea behind the proof follows quite closely Cantor's proof that the reals are uncountable (Theorem 2.5), and in fact the theorem can also be obtained fairly directly from that result (see Exercise 7.11). However, it is instructive to see the direct proof. The idea is to construct F^* in a way that will ensure that every possible machine M will in fact fail to compute F^*. We do so by defining $F^*(x)$ to equal 0 if x describes a Turing machine M which satisfies $M(x) = 1$ and defining $F^*(x) = 1$ otherwise. By construction, if M is any Turing machine and x is the string describing it, then $F^*(x) \neq M(x)$ and therefore M does *not* compute F^*.

⋆

Proof of Theorem 9.5. The proof is illustrated in Fig. 9.4. We start by defining the following function $G : \{0,1\}^* \to \{0,1\}$:

For every string $x \in \{0,1\}^*$, if x satisfies **(1)** x is a valid representation of some Turing machine M (per the representation scheme above) and **(2)** when the program M is executed on the input x it halts and produces an output, then we define $G(x)$ as the first bit of this output. Otherwise (i.e., if x is not a valid representation of a Turing machine, or the machine M_x never halts on x) we define $G(x) = 0$. We define $F^*(x) = 1 - G(x)$.

We claim that there is no Turing machine that computes F^*. Indeed, suppose, towards the sake of contradiction, there exists a machine M that computes F^*, and let x be the binary string that represents the machine M. On one hand, since by our assumption M computes F^*, on input x the machine M halts and outputs $F^*(x)$. On the other hand, by the definition of F^*, since x is the representation of the machine M, $F^*(x) = 1 - G(x) = 1 - M(x)$, hence yielding a contradiction.

■

Figure 9.4: We construct an uncomputable function by defining for every two strings x, y the value $1 - M_y(x)$ which equals 0 if the machine described by y outputs 1 on x, and 1 otherwise. We then define $F^*(x)$ to be the "diagonal" of this table, namely $F^*(x) = 1 - M_x(x)$ for every x. The function F^* is uncomputable, because if it was computable by some machine whose string description is x^* then we would get that $M_{x^*}(x^*) = F^*(x^*) = 1 - M_{x^*}(x^*)$.

Big Idea 12 There are some functions that *can not* be computed by *any* algorithm.

P

The proof of Theorem 9.5 is short but subtle. I suggest that you pause here and go back to read it again and think about it - this is a proof that is worth reading at least twice if not three or four times. It is not often the case that a few lines of mathematical reasoning establish a deeply profound fact - that there are problems we simply *cannot* solve.

The type of argument used to prove Theorem 9.5 is known as *diagonalization* since it can be described as defining a function based on the diagonal entries of a table as in Fig. 9.4. The proof can be thought of as an infinite version of the *counting* argument we used for showing lower bound for NAND-CIRC programs in Theorem 5.3. Namely, we show that it's not possible to compute all functions from $\{0,1\}^* \rightarrow \{0,1\}$ by Turing machines simply because there are more functions like that than there are Turing machines.

As mentioned in Remark 7.4, many texts use the "language" terminology and so will call a set $L \subseteq \{0,1\}^*$ an *undecidable* or *non-recursive* language if the function $F : \{0,1\}^* \rightarrow \{0,1\}$ such that $F(x) = 1 \leftrightarrow x \in L$ is uncomputable.

9.3 THE HALTING PROBLEM

Theorem 9.5 shows that there is *some* function that cannot be computed. But is this function the equivalent of the "tree that falls in the forest with no one hearing it"? That is, perhaps it is a function that no one actually *wants* to compute. It turns out that there are natural uncomputable functions:

Theorem 9.6 — Uncomputability of Halting function. Let $HALT : \{0,1\}^* \to \{0,1\}$ be the function such that for every string $M \in \{0,1\}^*$, $HALT(M,x) = 1$ if Turing machine M halts on the input x and $HALT(M,x) = 0$ otherwise. Then $HALT$ is not computable.

Before turning to prove Theorem 9.6, we note that $HALT$ is a very natural function to want to compute. For example, one can think of $HALT$ as a special case of the task of managing an "App store". That is, given the code of some application, the gatekeeper for the store needs to decide if this code is safe enough to allow in the store or not. At a minimum, it seems that we should verify that the code would not go into an infinite loop.

Proof Idea:

One way to think about this proof is as follows:

$$\text{Uncomputability of } F^* + \text{Universality} = \text{Uncomputability of } HALT$$

That is, we will use the universal Turing machine that computes $EVAL$ to derive the uncomputability of $HALT$ from the uncomputability of F^* shown in Theorem 9.5. Specifically, the proof will be by contradiction. That is, we will assume towards a contradiction that $HALT$ is computable, and use that assumption, together with the universal Turing machine of Theorem 9.1, to derive that F^* is computable, which will contradict Theorem 9.5.

⋆

Big Idea 13 If a function F is uncomputable we can show that another function H is uncomputable by giving a way to *reduce* the task of computing F to computing H.

Proof of Theorem 9.6. The proof will use the previously established result Theorem 9.5. Recall that Theorem 9.5 shows that the following function $F^* : \{0,1\}^* \to \{0,1\}$ is uncomputable:

$$F^*(x) = \begin{cases} 1 & x(x) = 0 \text{ or } x(x) = \perp \\ 0 & \text{otherwise} \end{cases}$$

where $x(x)$ denotes the output of the Turing machine described by the string x on the input x (with the usual convention that $x(x) = \perp$ if this computation does not halt).

We will show that the uncomputability of F^* implies the uncomputability of *HALT*. Specifically, we will assume, towards a contradiction, that there exists a Turing machine M that can compute the *HALT* function, and use that to obtain a Turing machine M' that computes the function F^*. (This is known as a proof by *reduction*, since we reduce the task of computing F^* to the task of computing *HALT*. By the contrapositive, this means the uncomputability of F^* implies the uncomputability of *HALT*.)

Indeed, suppose that M is a Turing machine that computes *HALT*. Algorithm 9.7 describes a Turing machine M' that computes F^*. (We use "high level" description of Turing machines, appealing to the "have your cake and eat it too" paradigm, see Big Idea 10.)

Algorithm 9.7 — F^* to $HALT$ reduction.

Input: $x \in \{0,1\}^*$
Output: $F^*(x)$

```
                          # Assume T.M. M_HALT computes HALT
Let z ← M_HALT(x, x).     # Assume z = HALT(x, x).
if z = 0 then
    return 1
end if
Let y ← U(x, x)           # U universal TM, i.e., y = x(x)
if y = 1 then
    return 0
end if
return 1
```

We claim that Algorithm 9.7 computes the function F^*. Indeed, suppose that $x(x) = 1$ (and hence $F^*(x) = 0$). In this case, $HALT(x,x) = 1$ and hence, under our assumption that $M(x,x) = HALT(x,x)$, the value z will equal 1, and hence Algorithm 9.7 will set $y = x(x) = 1$, and output the correct value 0.

Suppose otherwise that $x(x) \neq 0$. In this case there are two possibilities:

- **Case 1:** The machine described by x does not halt on the input x (and hence $F^*(x) = 1$). In this case, $HALT(x,x) = 0$. Since we assume that M computes *HALT* it means that on input x, x, the machine M must halt and output the value 0. This means that Algorithm 9.7 will set $z = 0$ and output 1.

- **Case 2:** The machine described by x halts on the input x and outputs some $y' \neq 0$ (and hence $F^*(x) = 0$). In this case, since $HALT(x,x) = 1$, under our assumptions, Algorithm 9.7 will set $y = y' \neq 0$ and so output 0.

We see that in all cases, $M'(x) = F^*(x)$, which contradicts the fact that F^* is uncomputable. Hence we reach a contradiction to our original assumption that M computes *HALT*.

■

Once again, this is a proof that's worth reading more than once. The uncomputability of the halting problem is one of the fundamental theorems of computer science, and is the starting point for much of the investigations we will see later. An excellent way to get a better understanding of Theorem 9.6 is to go over Section 9.3.2, which presents an alternative proof of the same result.

9.3.1 Is the Halting problem really hard? (discussion)

Many people's first instinct when they see the proof of Theorem 9.6 is to not believe it. That is, most people do believe the mathematical statement, but intuitively it doesn't seem that the Halting problem is really that hard. After all, being uncomputable only means that *HALT* cannot be computed by a Turing machine.

But programmers seem to solve *HALT* all the time by informally or formally arguing that their programs halt. It's true that their programs are written in C or Python, as opposed to Turing machines, but that makes no difference: we can easily translate back and forth between this model and any other programming language.

While every programmer encounters at some point an infinite loop, is there really no way to solve the halting problem? Some people argue that *they* personally can, if they think hard enough, determine whether any concrete program that they are given will halt or not. Some have even argued that humans in general have the ability to do that, and hence humans have inherently superior intelligence to computers or anything else modeled by Turing machines.[1]

The best answer we have so far is that there truly is no way to solve *HALT*, whether using Macs, PCs, quantum computers, humans, or any other combination of electronic, mechanical, and biological devices. Indeed this assertion is the content of the *Church-Turing Thesis*. This of course does not mean that for *every* possible program P, it is hard to decide if P enters an infinite loop. Some programs don't even have loops at all (and hence trivially halt), and there are many

[1] This argument has also been connected to the issues of consciousness and free will. I am personally skeptical of its relevance to these issues. Perhaps the reasoning is that humans have the ability to solve the halting problem but they exercise their free will and consciousness by choosing not to do so.

other far less trivial examples of programs that we can certify to never enter an infinite loop (or programs that we know for sure that *will* enter such a loop). However, there is no *general procedure* that would determine for an *arbitrary* program P whether it halts or not. Moreover, there are some very simple programs for which no one knows whether they halt or not. For example, the following Python program will halt if and only if Goldbach's conjecture is false:

```
def isprime(p):
    return all(p % i for i in range(2,p-1))

def Goldbach(n):
    return any( (isprime(p) and isprime(n-p))
           for p in range(2,n-1))

n = 4
while True:
    if not Goldbach(n): break
    n+= 2
```

Given that Goldbach's Conjecture has been open since 1742, it is unclear that humans have any magical ability to say whether this (or other similar programs) will halt or not.

9.3.2 A direct proof of the uncomputability of $HALT$ (optional)

It turns out that we can combine the ideas of the proofs of Theorem 9.5 and Theorem 9.6 to obtain a short proof of the latter theorem, that does not appeal to the uncomputability of F^*. This short proof appeared in print in a 1965 letter to the editor of Christopher Strachey:

> To the Editor, The Computer Journal.
>
> An Impossible Program
>
> Sir,
>
> A well-known piece of folk-lore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run. I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL, but not in any essential way.
>
> Suppose `T[R]` is a Boolean function taking a routine (or program) `R` with no formal or free variables as its arguments and that for all `R`, `T[R] = True` if `R` terminates if run and that `T[R] = False` if `R` does not terminate.

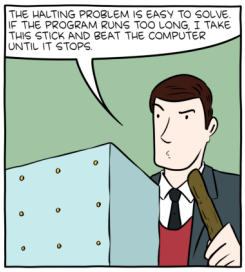


Figure 9.5: SMBC's take on solving the Halting problem.

Consider the routine P defined as follows

```
rec routine P
§L: if T[P] go to L
Return §
```

If `T[P] = True` the routine `P` will loop, and it will only terminate if `T[P] = False`. In each case `T[P]` has exactly the wrong value, and this contradiction shows that the function T cannot exist.

Yours faithfully,
C. Strachey

Churchill College, Cambridge

Try to stop and extract the argument for proving Theorem 9.6 from the letter above.

Since CPL is not as common today, let us reproduce this proof. The idea is the following: suppose for the sake of contradiction that there exists a program `T` such that `T(f,x)` equals `True` iff `f` halts on input `x`. (Strachey's letter considers the no-input variant of *HALT*, but as we'll see, this is an immaterial distinction.) Then we can construct a program `P` and an input `x` such that `T(P,x)` gives the wrong answer. The idea is that on input `x`, the program `P` will do the following: run `T(x,x)`, and if the answer is `True` then go into an infinite loop, and otherwise halt. Now you can see that `T(P,P)` will give the wrong answer: if `P` halts when it gets its own code as input, then `T(P,P)` is supposed to be `True`, but then `P(P)` will go into an infinite loop. And if `P` does not halt, then `T(P,P)` is supposed to be `False` but then `P(P)` will halt. We can also code this up in Python:

```
def CantSolveMe(T):
    """
    Gets function T that claims to solve HALT.
    Returns a pair (P,x) of code and input on which
    T(P,x) ≠ HALT(x)
    """
    def fool(x):
        if T(x,x):
            while True: pass
        return "I halted"

    return (fool,fool)
```

For example, consider the following Naive Python program `T` that guesses that a given function does not halt if its input contains `while` or `for`

```
def T(f,x):
    """Crude halting tester - decides it doesn't halt if it
     ↪  contains a loop."""
    import inspect
    source = inspect.getsource(f)
    if source.find("while"): return False
    if source.find("for"): return False
    return True
```

If we now set `(f,x) = CantSolveMe(T)`, then `T(f,x)=False` but `f(x)` does in fact halt. This is of course not specific to this particular `T`: for every program `T`, if we run `(f,x) = CantSolveMe(T)` then we'll get an input on which `T` gives the wrong answer to *HALT*.

9.4 REDUCTIONS

The Halting problem turns out to be a linchpin of uncomputability, in the sense that Theorem 9.6 has been used to show the uncomputability of a great many interesting functions. We will see several examples of such results in this chapter and the exercises, but there are many more such results (see Fig. 9.6).

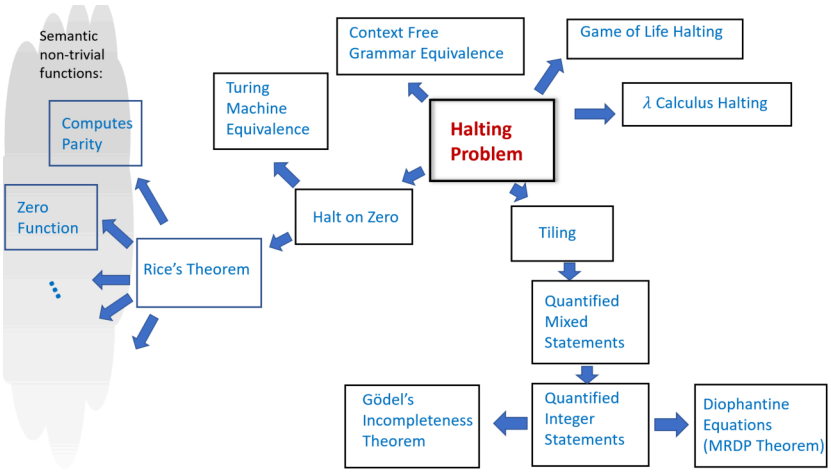


Figure 9.6: Some uncomputability results. An arrow from problem X to problem Y means that we use the uncomputability of X to prove the uncomputability of Y by reducing computing X to computing Y. All of these results except for the MRDP Theorem appear in either the text or exercises. The Halting Problem *HALT* serves as our starting point for all these uncomputability results as well as many others.

The idea behind such uncomputability results is conceptually simple but can at first be quite confusing. If we know that *HALT* is uncomputable, and we want to show that some other function *BLAH* is uncomputable, then we can do so via a *contrapositive* argument (i.e., proof by contradiction). That is, we show that **if** there exists a Turing machine that computes *BLAH* **then** there exists a Turing machine that computes *HALT*. (Indeed, this is exactly how we showed that *HALT* itself is uncomputable, by deriving this fact from the uncomputability of the function F^* of Theorem 9.5.)

For example, to prove that *BLAH* is uncomputable, we could show that there is a computable function $R : \{0,1\}^* \to \{0,1\}^*$ such that for every pair M and x, $HALT(M,x) = BLAH(R(M,x))$. The existence of such a function R implies that **if** *BLAH* was computable **then** *HALT* would be computable as well, hence leading to a contradiction! The confusing part about reductions is that we are assuming something we *believe* is false (that *BLAH* has an algorithm) to derive something that we *know* is false (that *HALT* has an algorithm). Michael Sipser describes such results as having the form *"If pigs could whistle then horses could fly"*.

A reduction-based proof has two components. For starters, since we need R to be computable, we should describe the algorithm to compute it. The algorithm to compute R is known as a *reduction* since the transformation R modifies an input to *HALT* to an input to *BLAH*, and hence *reduces* the task of computing *HALT* to the task of computing *BLAH*. The second component of a reduction-based proof is the *analysis* of the algorithm R: namely a proof that R does indeed satisfy the desired properties.

Reduction-based proofs are just like other proofs by contradiction, but the fact that they involve hypothetical algorithms that don't really exist tends to make reductions quite confusing. The one silver lining is that at the end of the day the notion of reductions is mathematically quite simple, and so it's not that bad even if you have to go back to first principles every time you need to remember what is the direction that a reduction should go in.

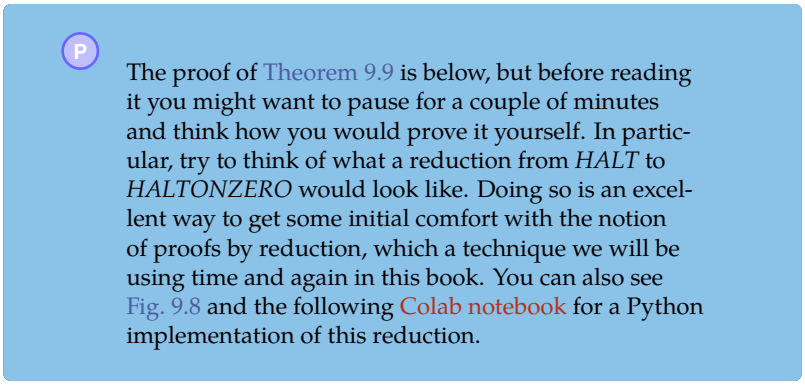
Remark 9.8 — Reductions are algorithms. A reduction is an *algorithm*, which means that, as discussed in Remark 0.3, a reduction has three components:

- **Specification (what):** In the case of a reduction from *HALT* to *BLAH*, the specification is that function $R : \{0,1\}^* \to \{0,1\}^*$ should satisfy that $HALT(M,x) = BLAH(R(M,x))$ for every Turing machine M and input x. In general, to reduce a function F to G, the reduction should satisfy $F(w) = G(R(w))$ for every input w to F.
- **Implementation (how):** The algorithm's description: the precise instructions how to transform an input w to the output $y = R(w)$.
- **Analysis (why):** A *proof* that the algorithm meets the specification. In particular, in a reduction from F to G this is a proof that for every input w, the output $y = R(w)$ of the algorithm satisfies that $F(w) = G(y)$.

9.4.1 Example: Halting on the zero problem

Here is a concrete example for a proof by reduction. We define the function $HALTONZERO : \{0,1\}^* \to \{0,1\}$ as follows. Given any string M, $HALTONZERO(M) = 1$ if and only if M describes a Turing machine that halts when it is given the string 0 as input. A priori $HALTONZERO$ seems like a potentially easier function to compute than the full-fledged $HALT$ function, and so we could perhaps hope that it is not uncomputable. Alas, the following theorem shows that this is not the case:

Theorem 9.9 — Halting without input. $HALTONZERO$ is uncomputable.



The proof of Theorem 9.9 is below, but before reading it you might want to pause for a couple of minutes and think how you would prove it yourself. In particular, try to think of what a reduction from $HALT$ to $HALTONZERO$ would look like. Doing so is an excellent way to get some initial comfort with the notion of proofs by reduction, which a technique we will be using time and again in this book. You can also see Fig. 9.8 and the following Colab notebook for a Python implementation of this reduction.

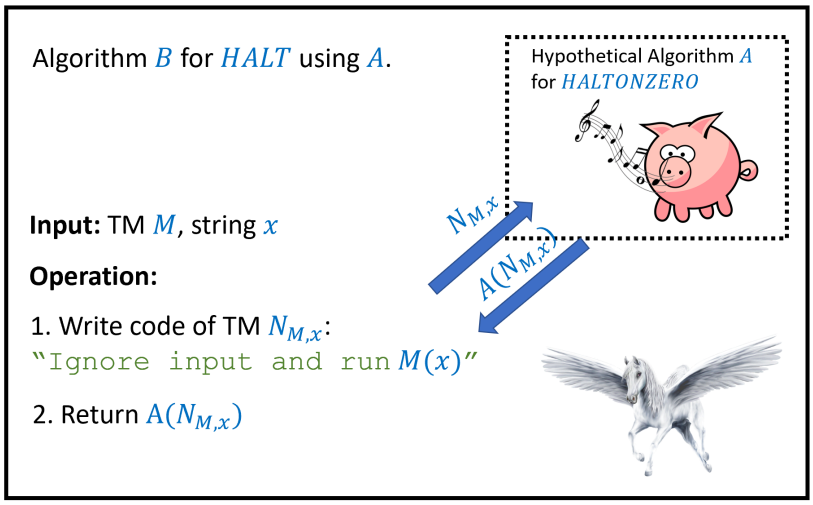


Figure 9.7: To prove Theorem 9.9, we show that $HALTONZERO$ is uncomputable by giving a *reduction* from the task of computing $HALT$ to the task of computing $HALTONZERO$. This shows that if there was a hypothetical algorithm A computing $HALTONZERO$, then there would be an algorithm B computing $HALT$, contradicting Theorem 9.6. Since neither A nor B actually exists, this is an example of an implication of the form "if pigs could whistle then horses could fly".

Proof of Theorem 9.9. The proof is by reduction from $HALT$, see Fig. 9.7. We will assume, towards the sake of contradiction, that $HALTONZERO$ is computable by some algorithm A, and use this hypothetical algorithm A to construct an algorithm B to compute

HALT, hence obtaining a contradiction to Theorem 9.6. (As discussed in Big Idea 10, following our "have your cake and eat it too" paradigm, we just use the generic name "algorithm" rather than worrying whether we model them as Turing machines, NAND-TM programs, NAND-RAM, etc.; this makes no difference since all these models are equivalent to one another.)

Since this is our first proof by reduction from the Halting problem, we will spell it out in more details than usual. Such a proof by reduction consists of two steps:

1. *Description of the reduction:* We will describe the operation of our algorithm B, and how it makes "function calls" to the hypothetical algorithm A.

2. *Analysis of the reduction:* We will then prove that under the hypothesis that Algorithm A computes *HALTONZERO*, Algorithm B will compute *HALT*.

Algorithm 9.10 — $HALT$ to $HALTONZERO$ reduction.

Input: Turing machine M and string x.
Output: Turing machine M' such that M halts on x iff M' halts on zero

procedure $N_{M,x}(w)$ # *Description of the T.M.* $N_{M,x}$
    return $EVAL(M,x)$ # *Ignore the* ***Input:*** w*, evaluate* M *on* x*.*
end procedure
return $N_{M,x}$ # *We do not execute* $N_{M,x}$*: only* ***return*** *its description*

Our Algorithm B works as follows: on input M, x, it runs Algorithm 9.10 to obtain a Turing machine M', and then returns $A(M')$. The machine M' ignores its input z and simply runs M on x.

In pseudocode, the program $N_{M,x}$ will look something like the following:

```
def N(z):
    M = r'.......'
    # a string constant containing desc. of M
    x = r'.......'
    # a string constant containing x
    return eval(M,x)
    # note that we ignore the input z
```

That is, if we think of $N_{M,x}$ as a program, then it is a program that contains M and x as "hardwired constants", and given any input z, it

simply ignores the input and always returns the result of evaluating M on x. The algorithm B does *not* actually execute the machine $N_{M,x}$. B merely writes down the description of $N_{M,x}$ as a string (just as we did above) and feeds this string as input to A.

The above completes the *description* of the reduction. The *analysis* is obtained by proving the following claim:

Claim: For every strings M, x, z, the machine $N_{M,x}$ constructed by Algorithm B in Step 1 satisfies that $N_{M,x}$ halts on z if and only if the program described by M halts on the input x.

Proof of Claim: Since $N_{M,x}$ ignores its input and evaluates M on x using the universal Turing machine, it will halt on z if and only if M halts on x.

In particular if we instantiate this claim with the input $z = 0$ to $N_{M,x}$, we see that $HALTONZERO(N_{M,x}) = HALT(M, x)$. Thus if the hypothetical algorithm A satisfies $A(M) = HALTONZERO(M)$ for every M then the algorithm B we construct satisfies $B(M, x) = HALT(M, x)$ for every M, x, contradicting the uncomputability of $HALT$.

■

```python
def B(P,x):
  """B will solve the Halting problem
  if A solves the HALTONZERO problem
  INPUT:
    P: source code of Python function
    x: input to P
  USES: Black box A(·)
  If ∀Q A(Q)=HALTONZERO(Q) then will return HALT(P,x)"""

  # extract name of function defined in P
  i,j  = P.index("def"), P.index("(")
  func = P[i+3:j]

  Q_x = f"def Q(z):\n {func}('{x}')\n"+P

  return A(Q_x)
```

Figure 9.8: A Python implementation of the reduction showing that *HALTONZERO* is uncomputable if *HALT* is. See this Colab notebook for a full implementation of the reduction.

R

Remark 9.11 — The hardwiring technique. In the proof of Theorem 9.9 we used the technique of "hardwiring" an input x to a program/machine P. That is, we take a program that computes the function $x \mapsto f(x)$ and "fix" or "hardwire" some of the inputs to some constant value. For example, if you have a program that takes as input a pair of numbers x, y and outputs their product (i.e., computes the function $f(x, y) = x \times y$),

then you can "hardwire" the second input to be 17 and obtain a program that takes as input a number x and outputs $x \times 17$ (i.e., computes the function $g(x) = x \times 17$). This technique is quite common in reductions and elsewhere, and we will use it time and again in this book.

9.5 RICE'S THEOREM AND THE IMPOSSIBILITY OF GENERAL SOFTWARE VERIFICATION

The uncomputability of the Halting problem turns out to be a special case of a much more general phenomenon. Namely, that *we cannot certify semantic properties of general purpose programs*. "Semantic properties" mean properties of the *function* that the program computes, as opposed to properties that depend on the particular syntax used by the program.

An example for a *semantic property* of a program P is the property that whenever P is given an input string with an even number of 1's, it outputs 0. Another example is the property that P will always halt whenever the input ends with a 1. In contrast, the property that a C program contains a comment before every function declaration is not a semantic property, since it depends on the actual source code as opposed to the input/output relation.

Checking semantic properties of programs is of great interest, as it corresponds to checking whether a program conforms to a specification. Alas it turns out that such properties are in general *uncomputable*. We have already seen some examples of uncomputable semantic functions, namely *HALT* and *HALTONZERO*, but these are just the "tip of the iceberg". We start by observing one more such example:

Theorem 9.12 — Computing all zero function. Let $ZEROFUNC : \{0,1\}^* \to \{0,1\}$ be the function such that for every $M \in \{0,1\}^*$, $ZEROFUNC(M) = 1$ if and only if M represents a Turing machine such that M outputs 0 on every input $x \in \{0,1\}^*$. Then *ZEROFUNC* is uncomputable.

Despite the similarity in their names, *ZEROFUNC* and *HALTONZERO* are two different functions. For example, if M is a Turing machine that on input $x \in \{0,1\}^*$, halts and outputs the OR of all of x's coordinates, then $HALTONZERO(M) = 1$ (since M does halt on the input 0) but $ZEROFUNC(M) = 0$ (since M does not compute the constant zero function).

Proof of Theorem 9.12. The proof is by reduction from *HALTONZERO*. Suppose, towards the sake of contradiction, that there was an algorithm A such that $A(M) = \mathit{ZEROFUNC}(M)$ for every $M \in \{0,1\}^*$. Then we will construct an algorithm B that solves *HALTONZERO*, contradicting Theorem 9.9.

Given a Turing machine N (which is the input to *HALTONZERO*), our Algorithm B does the following:

1. Construct a Turing machine M which on input $x \in \{0,1\}^*$, first runs $N(0)$ and then outputs 0.
2. Return $A(M)$.

Now if N halts on the input 0 then the Turing machine M computes the constant zero function, and hence under our assumption that A computes *ZEROFUNC*, $A(M) = 1$. If N does not halt on the input 0, then the Turing machine M will not halt on any input, and so in particular will *not* compute the constant zero function. Hence under our assumption that A computes *ZEROFUNC*, $A(M) = 0$. We see that in both cases, $\mathit{ZEROFUNC}(M) = \mathit{HALTONZERO}(N)$ and hence the value that Algorithm B returns in step 2 is equal to $\mathit{HALTONZERO}(N)$ which is what we needed to prove. ■

Another result along similar lines is the following:

Theorem 9.13 — Uncomputability of verifying parity. The following function is uncomputable

$$\mathit{COMPUTES\text{-}PARITY}(P) = \begin{cases} 1 & P \text{ computes the parity function} \\ 0 & \text{otherwise} \end{cases}$$

We leave the proof of Theorem 9.13 as an exercise (Exercise 9.6). I strongly encourage you to stop here and try to solve this exercise.

9.5.1 Rice's Theorem

Theorem 9.13 can be generalized far beyond the parity function. In fact, this generalization rules out verifying any type of semantic specification on programs. We define a *semantic specification* on programs to be some property that does not depend on the code of the program but just on the function that the program computes.

For example, consider the following two C programs

```
int First(int n) {
    if (n<0) return 0;
    return 2*n;
}

int Second(int n) {
    int i = 0;
    int j = 0
    if (n<0) return 0;
    while (j<n) {
        i = i + 2;
        j = j + 1;
    }
    return i;
}
```

`First` and `Second` are two distinct C programs, but they compute the same function. A *semantic* property, would be either *true* for both programs or *false* for both programs, since it depends on the *function* the programs compute and not on their code. An example for a semantic property that both `First` and `Second` satisfy is the following: *"The program P computes a function f mapping integers to integers satisfying that $f(n) \geq n$ for every input n".*

A property is *not semantic* if it depends on the *source code* rather than the input/output behavior. For example, properties such as "the program contains the variable `k`" or "the program uses the `while` operation" are not semantic. Such properties can be true for one of the programs and false for others. Formally, we define semantic properties as follows:

Definition 9.14 — Semantic properties. A pair of Turing machines M and M' are *functionally equivalent* if for every $x \in \{0,1\}^*$, $M(x) = M'(x)$. (In particular, $M(x) = \perp$ iff $M'(x) = \perp$ for all x.)

A function $F : \{0,1\}^* \to \{0,1\}$ is *semantic* if for every pair of strings M, M' that represent functionally equivalent Turing machines, $F(M) = F(M')$. (Recall that we assume that every string represents *some* Turing machine, see Remark 9.3)

There are two trivial examples of semantic functions: the constant one function and the constant zero function. For example, if Z is the constant zero function (i.e., $Z(M) = 0$ for every M) then clearly $F(M) = F(M')$ for every pair of Turing machines M and M' that are functionally equivalent M and M'. Here is a non-trivial example

Solved Exercise 9.1 — $ZEROFUNC$ is semantic. Prove that the function *ZEROFUNC* is semantic.

■

Solution:

Recall that $ZEROFUNC(M) = 1$ if and only if $M(x) = 0$ for every $x \in \{0,1\}^*$. If M and M' are functionally equivalent, then for every x, $M(x) = M'(x)$. Hence $ZEROFUNC(M) = 1$ if and only if $ZEROFUNC(M') = 1$.

■

Often the properties of programs that we are most interested in computing are the *semantic* ones, since we want to understand the programs' functionality. Unfortunately, Rice's Theorem tells us that these properties are all uncomputable:

Theorem 9.15 — Rice's Theorem. Let $F : \{0,1\}^* \to \{0,1\}$. If F is semantic and non-trivial then it is uncomputable.

Proof Idea:

The idea behind the proof is to show that every semantic non-trivial function F is at least as hard to compute as *HALTONZERO*. This will conclude the proof since by Theorem 9.9, *HALTONZERO* is uncomputable. If a function F is non-trivial then there are two machines M_0 and M_1 such that $F(M_0) = 0$ and $F(M_1) = 1$. So, the goal would be to take a machine N and find a way to map it into a machine $M = R(N)$, such that **(i)** if N halts on zero then M is functionally equivalent to M_1 and **(ii)** if N does *not* halt on zero then M is functionally equivalent to M_0.

Because F is semantic, if we achieved this, then we would be guaranteed that $HALTONZERO(N) = F(R(N))$, and hence would show that if F was computable, then *HALTONZERO* would be computable as well, contradicting Theorem 9.9.

⋆

Proof of Theorem 9.15. We will not give the proof in full formality, but rather illustrate the proof idea by restricting our attention to a particular semantic function F. However, the same techniques generalize to all possible semantic functions. Define $MONOTONE : \{0,1\}^* \to \{0,1\}$ as follows: $MONOTONE(M) = 1$ if there does not exist $n \in \mathbb{N}$ and two inputs $x, x' \in \{0,1\}^n$ such that for every $i \in [n]$ $x_i \leq x'_i$ but $M(x)$ outputs 1 and $M(x') = 0$. That is, $MONOTONE(M) = 1$ if it's not possible to find an input x such that flipping some bits of x from 0 to 1 will change M's output in the other direction from 1 to 0. We will

prove that *MONOTONE* is uncomputable, but the proof will easily generalize to any semantic function.

We start by noting that *MONOTONE* is neither the constant zero nor the constant one function:

- The machine *INF* that simply goes into an infinite loop on every input satisfies $MONOTONE(INF) = 1$, since *INF* is not defined *anywhere* and so in particular there are no two inputs x, x' where $x_i \leq x'_i$ for every i but $INF(x) = 0$ and $INF(x') = 1$.

- The machine *PAR* that computes the XOR or parity of its input, is not monotone (e.g., $PAR(1,1,0,0,\ldots,0) = 0$ but $PAR(1,0,0,\ldots,0) = 0$) and hence $MONOTONE(PAR) = 0$.

(Note that *INF* and *PAR* are *machines* and not *functions*.)

We will now give a reduction from *HALTONZERO* to *MONOTONE*. That is, we assume towards a contradiction that there exists an algorithm A that computes *MONOTONE* and we will build an algorithm B that computes *HALTONZERO*. Our algorithm B will work as follows:

Algorithm B:

Input: String N describing a Turing machine. (*Goal:* Compute $HALTONZERO(N)$)

Assumption: Access to Algorithm A to compute *MONOTONE*.

Operation:

1. Construct the following machine M: "On input $z \in \{0,1\}^*$ do: **(a)** Run $N(0)$, **(b)** Return $PAR(z)$".
2. Return $1 - A(M)$.

To complete the proof we need to show that B outputs the correct answer, under our assumption that A computes *MONOTONE*. In other words, we need to show that $HALTONZERO(N) = 1 - MONOTONE(M)$. Suppose that N does *not* halt on zero. In this case the program M constructed by Algorithm B enters into an infinite loop in step **(a)** and will never reach step **(b)**. Hence in this case N is functionally equivalent to *INF*. (The machine N is not the same machine as *INF*: its description or *code* is different. But it does have the same input/output behavior (in this case) of never halting on any input. Also, while the program M will go into an infinite loop on every input, Algorithm B never actually runs M: it only produces its code and feeds it to A. Hence Algorithm B will *not* enter into an infinite loop even in this case.) Thus in this case, $MONOTONE(M) = MONOTONE(INF) = 1$.

If N *does* halt on zero, then step **(a)** in M will eventually conclude and M's output will be determined by step **(b)**, where it simply outputs the parity of its input. Hence in this case, M computes the non-monotone parity function (i.e., is functionally equivalent to PAR), and so we get that $MONOTONE(M) = MONOTONE(PAR) = 0$. In both cases, $MONOTONE(M) = 1 - HALTONZERO(N)$, which is what we wanted to prove.

An examination of this proof shows that we did not use anything about *MONOTONE* beyond the fact that it is semantic and non-trivial. For every semantic non-trivial F, we can use the same proof, replacing *PAR* and *INF* with two machines M_0 and M_1 such that $F(M_0) = 0$ and $F(M_1) = 1$. Such machines must exist if F is non-trivial.

■

R

Remark 9.16 — Semantic is not the same as uncomputable. Rice's Theorem is so powerful and such a popular way of proving uncomputability that people sometimes get confused and think that it is the *only* way to prove uncomputability. In particular, a common misconception is that if a function F is *not* semantic then it is *computable*. This is not at all the case.

For example, consider the following function $HALTNOYALE : \{0,1\}^* \to \{0,1\}$. This is a function that on input a string that represents a NAND-TM program P, outputs 1 if and only if both **(i)** P halts on the input 0, and **(ii)** the program P does not contain a variable with the identifier `Yale`. The function *HALTNOYALE* is clearly not semantic, as it will output two different values when given as input one of the following two functionally equivalent programs:

```
Yale[0] = NAND(X[0],X[0])
Y[0] = NAND(X[0],Yale[0])
```

and

```
Harvard[0] = NAND(X[0],X[0])
Y[0] = NAND(X[0],Harvard[0])
```

However, *HALTNOYALE* is uncomputable since every program P can be transformed into an equivalent (and in fact improved `:)`) program P' that does not contain the variable `Yale`. Hence if we could compute *HALTNOYALE* then determine halting on zero for NAND-TM programs (and hence for Turing machines as well).

Moreover, as we will see in Chapter 11, there are uncomputable functions whose inputs are not programs, and hence for which the adjective "semantic" is not applicable.

Properties such as "the program contains the variable `Yale`" are sometimes known as *syntactic* properties. The terms "semantic" and "syntactic" are used beyond the realm of programming languages: a famous example of a syntactically correct but semantically meaningless sentence in English is Chomsky's "Colorless green ideas sleep furiously." However, formally defining "syntactic properties" is rather subtle and we will not use this terminology in this book, sticking to the terms "semantic" and "non-semantic" only.

9.5.2 Halting and Rice's Theorem for other Turing-complete models

As we saw before, many natural computational models turn out to be *equivalent* to one another, in the sense that we can transform a "program" of one model (such as a λ expression, or a game-of-life configurations) into another model (such as a NAND-TM program). This equivalence implies that we can translate the uncomputability of the Halting problem for NAND-TM programs into uncomputability for Halting in other models. For example:

Theorem 9.17 — NAND-TM Machine Halting. Let $NANDTMHALT : \{0,1\}^* \to \{0,1\}$ be the function that on input strings $P \in \{0,1\}^*$ and $x \in \{0,1\}^*$ outputs 1 if the NAND-TM program described by P halts on the input x and outputs 0 otherwise. Then $NANDTMHALT$ is uncomputable.

P Once again, this is a good point for you to stop and try to prove the result yourself before reading the proof below.

Proof. We have seen in Theorem 7.11 that for every Turing machine M, there is an equivalent NAND-TM program P_M such that for every x, $P_M(x) = M(x)$. In particular this means that $HALT(M) = NANDTMHALT(P_M)$.

The transformation $M \mapsto P_M$ that is obtained from the proof of Theorem 7.11 is *constructive*. That is, the proof yields a way to *compute* the map $M \mapsto P_M$. This means that this proof yields a *reduction* from task of computing $HALT$ to the task of computing $NANDTMHALT$, which means that since $HALT$ is uncomputable, neither is $NANDTMHALT$.

■

The same proof carries over to other computational models such as the λ *calculus*, *two dimensional* (or even one-dimensional) *automata* etc. Hence for example, there is no algorithm to decide if a λ expression evaluates the identity function, and no algorithm to decide whether an initial configuration of the game of life will result in eventually coloring the cell $(0,0)$ black or not.

Indeed, we can generalize Rice's Theorem to all these models. For example, if $F : \{0,1\}^* \rightarrow \{0,1\}$ is a non-trivial function such that $F(P) = F(P')$ for every functionally equivalent NAND-TM programs P, P' then F is uncomputable, and the same holds for NAND-RAM programs, λ-expressions, and all other Turing complete models (as defined in Definition 8.5), see also Exercise 9.12.

9.5.3 Is software verification doomed? (discussion)

Programs are increasingly being used for mission critical purposes, whether it's running our banking system, flying planes, or monitoring nuclear reactors. If we can't even give a certification algorithm that a program correctly computes the parity function, how can we ever be assured that a program does what it is supposed to do? The key insight is that while it is impossible to certify that a *general* program conforms with a specification, it is possible to write a program in the first place in a way that will make it easier to certify. As a trivial example, if you write a program without loops, then you can certify that it halts. Also, while it might not be possible to certify that an *arbitrary* program computes the parity function, it is quite possible to write a particular program P for which we can mathematically *prove* that P computes the parity. In fact, writing programs or algorithms and providing proofs for their correctness is what we do all the time in algorithms research.

The field of *software verification* is concerned with verifying that given programs satisfy certain conditions. These conditions can be that the program computes a certain function, that it never writes into a dangerous memory location, that is respects certain invariants, and others. While the general tasks of verifying this may be uncomputable, researchers have managed to do so for many interesting cases, especially if the program is written in the first place in a formalism or programming language that makes verification easier. That said, verification, especially of large and complex programs, remains a highly challenging task in practice as well, and the number of programs that have been formally proven correct is still quite small. Moreover, even phrasing the right theorem to prove (i.e., the specification) is often a highly non-trivial endeavor.

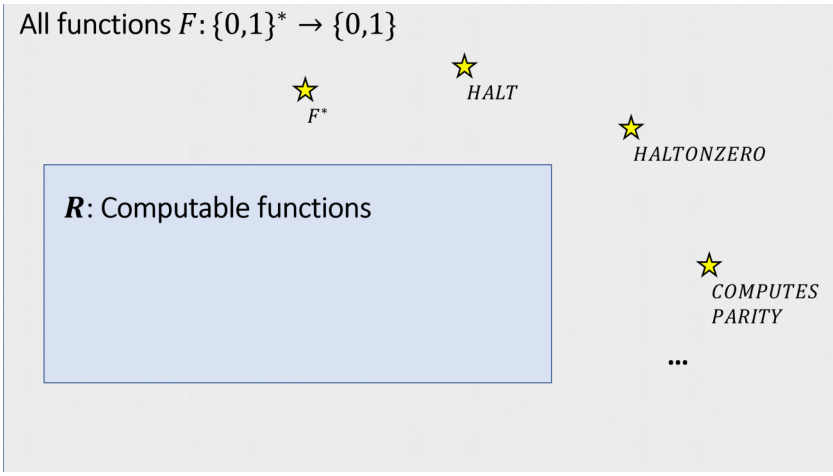


Figure 9.9: The set **R** of computable Boolean functions (Definition 7.3) is a proper subset of the set of all functions mapping $\{0,1\}^*$ to $\{0,1\}$. In this chapter we saw a few examples of elements in the latter set that are not in the former.

Chapter Recap

- There is a *universal* Turing machine (or NAND-TM program) U such that on input a description of a Turing machine M and some input x, $U(M,x)$ halts and outputs $M(x)$ if (and only if) M halts on input x. Unlike in the case of finite computation (i.e., NAND-CIRC programs / circuits), the input to the program U can be a machine M that has more states than U itself.
- Unlike the finite case, there are actually functions that are *inherently uncomputable* in the sense that they cannot be computed by *any* Turing machine.
- These include not only some "degenerate" or "esoteric" functions but also functions that people have deeply care about and conjectured that could be computed.
- If the Church-Turing thesis holds then a function F that is uncomputable according to our definition cannot be computed by any means in our physical world.

9.6 EXERCISES

Exercise 9.1 — NAND-RAM Halt. Let $NANDRAMHALT : \{0,1\}^* \to \{0,1\}$ be the function such that on input (P,x) where P represents a NAND-RAM program, $NANDRAMHALT(P,x) = 1$ iff P halts on the input x. Prove that $NANDRAMHALT$ is uncomputable.

Exercise 9.2 — Timed halting. Let $TIMEDHALT : \{0,1\}^* \to \{0,1\}$ be the function that on input (a string representing) a triple (M,x,T),

$TIMEDHALT(M, x, T) = 1$ iff the Turing machine M, on input x, halts within at most T steps (where a *step* is defined as one sequence of reading a symbol from the tape, updating the state, writing a new symbol and (potentially) moving the head).

Prove that *TIMEDHALT* is *computable*.

■

Exercise 9.3 — Space halting (challenging). Let $SPACEHALT : \{0,1\}^* \to \{0,1\}$ be the function that on input (a string representing) a triple (M, x, T), $SPACEHALT(M, x, T) = 1$ iff the Turing machine M, on input x, halts before its head reached the T-th location of its tape. (We don't care how many steps M makes, as long as the head stays inside locations $\{0, \ldots, T-1\}$.)

Prove that *SPACEHALT* is *computable*. See footnote for hint[2]

■

Exercise 9.4 — Computable compositions. Suppose that $F : \{0,1\}^* \to \{0,1\}$ and $G : \{0,1\}^* \to \{0,1\}$ are computable functions. For each one of the following functions H, either prove that H is *necessarily computable* or give an example of a pair F and G of computable functions such that H will not be computable. Prove your assertions.

1. $H(x) = 1$ iff $F(x) = 1$ OR $G(x) = 1$.
2. $H(x) = 1$ iff there exist two non-empty strings $u, v \in \{0,1\}^*$ such that $x = uv$ (i.e., x is the concatenation of u and v), $F(u) = 1$ and $G(v) = 1$.
3. $H(x) = 1$ iff there exist a list $u_0, \ldots, u_{t-1}$ of non-empty strings such that strings$F(u_i) = 1$ for every $i \in [t]$ and $x = u_0 u_1 \cdots u_{t-1}$.
4. $H(x) = 1$ iff x is a valid string representation of a NAND++ program P such that for every $z \in \{0,1\}^*$, on input z the program P outputs $F(z)$.
5. $H(x) = 1$ iff x is a valid string representation of a NAND++ program P such that on input x the program P outputs $F(x)$.
6. $H(x) = 1$ iff x is a valid string representation of a NAND++ program P such that on input x, P outputs $F(x)$ after executing at most $100 \cdot |x|^2$ lines.

■

Exercise 9.5 Prove that the following function $FINITE : \{0,1\}^* \to \{0,1\}$ is uncomputable. On input $P \in \{0,1\}^*$, we define $FINITE(P) = 1$ if and only if P is a string that represents a NAND++ program such that there only a finite number of inputs $x \in \{0,1\}^*$ s.t. $P(x) = 1$.[3]

■

[2] A machine with alphabet Σ can have at most $|\Sigma|^T$ choices for the contents of the first T locations of its tape. What happens if the machine repeats a previously seen configuration, in the sense that the tape contents, the head location, and the current state, are all identical to what they were in some previous state of the execution?

[3] Hint: You can use Rice's Theorem.

Exercise 9.6 — Computing parity. Prove Theorem 9.13 without using Rice's Theorem.

■

Exercise 9.7 — TM Equivalence. Let $EQ : \{0,1\}^* :\to \{0,1\}$ be the function defined as follows: given a string representing a pair (M, M') of Turing machines, $EQ(M, M') = 1$ iff M and M' are functionally equivalent as per Definition 9.14. Prove that EQ is uncomputable.

Note that you *cannot* use Rice's Theorem directly, as this theorem only deals with functions that take a single Turing machine as input, and EQ takes two machines.

■

Exercise 9.8 For each of the following two functions, say whether it is computable or not:

1. Given a NAND-TM program P, an input x, and a number k, when we run P on x, does the index variable `i` ever reach k?
2. Given a NAND-TM program P, an input x, and a number k, when we run P on x, does P ever write to an array at index k?

■

Exercise 9.9 Let $F : \{0,1\}^* \to \{0,1\}$ be the function that is defined as follows. On input a string P that represents a NAND-RAM program and a String M that represents a Turing machine, $F(P, M) = 1$ if and only if there exists some input x such P halts on x but M does not halt on x. Prove that F is uncomputable. See footnote for hint.[4]

■

Exercise 9.10 — Recursively enumerable. Define a function $F : \{0,1\}^* :\to \{0,1\}$ to be *recursively enumerable* if there exists a Turing machine M such that such that for every $x \in \{0,1\}^*$, if $F(x) = 1$ then $M(x) = 1$, and if $F(x) = 0$ then $M(x) = \bot$. (i.e., if $F(x) = 0$ then M does not halt on x.)

1. Prove that every computable F is also recursively enumerable.
2. Prove that there exists F that is not computable but is recursively enumerable. See footnote for hint.[5]
3. Prove that there exists a function $F : \{0,1\}^* \to \{0,1\}$ such that F is not recursively enumerable. See footnote for hint.[6]
4. Prove that there exists a function $F : \{0,1\}^* \to \{0,1\}$ such that F is recursively enumerable but the function $\overline{F}$ defined as $\overline{F}(x) = 1 - F(x)$ is *not* recursively enumerable. See footnote for hint.[7]

■

[4] *Hint:* While it cannot be applied directly, with a little "massaging" you can prove this using Rice's Theorem.

[5] *HALT* has this property.

[6] You can either use the diagonalization method to prove this directly or show that the set of all recursively enumerable functions is *countable*.

[7] *HALT* has this property: show that if both *HALT* and $\overline{HALT}$ were recursively enumerable then *HALT* would be in fact computable.

Exercise 9.11 — Rice's Theorem: standard form. In this exercise we will prove Rice's Theorem in the form that it is typically stated in the literature.

For a Turing machine M, define $L(M) \subseteq \{0,1\}^*$ to be the set of all $x \in \{0,1\}^*$ such that M halts on the input x and outputs 1. (The set $L(M)$ is known in the literature as the *language recognized by* M. Note that M might either output a value other than 1 or not halt at all on inputs $x \notin L(M)$.)

1. Prove that for every Turing machine M, if we define $F_M : \{0,1\}^* \to \{0,1\}$ to be the function such that $F_M(x) = 1$ iff $x \in L(M)$ then F_M is *recursively enumerable* as defined in Exercise 9.10.

2. Use Theorem 9.15 to prove that for every $G : \{0,1\}^* \to \{0,1\}$, if **(a)** G is neither the constant zero nor the constant one function, and **(b)** for every M, M' such that $L(M) = L(M')$, $G(M) = G(M')$, then G is uncomputable. See footnote for hint.[8]

[8] Show that any G satisfying **(b)** must be semantic.

■

Exercise 9.12 — Rice's Theorem for general Turing-equivalent models (optional). Let $\mathcal{F}$ be the set of all partial functions from $\{0,1\}^*$ to $\{0,1\}$ and $\mathcal{M} : \{0,1\}^* \to \mathcal{F}$ be a Turing-equivalent model as defined in Definition 8.5. We define a function $F : \{0,1\}^* \to \{0,1\}$ to be *$\mathcal{M}$-semantic* if there exists some $\mathcal{G} : \mathcal{F} \to \{0,1\}$ such that $F(P) = \mathcal{G}(\mathcal{M}(P))$ for every $P \in \{0,1\}^*$.

Prove that for every $\mathcal{M}$-semantic $F : \{0,1\}^* \to \{0,1\}$ that is neither the constant one nor the constant zero function, F is uncomputable.

■

Exercise 9.13 — Busy Beaver. In this question we define the NAND-TM variant of the busy beaver function (see Aaronson's 1999 essay, 2017 blog post and 2020 survey [Aar20]; see also Tao's highly recommended presentation on how civilization's scientific progress can be measured by the quantities we can grasp).

1. Let $T_{BB} : \{0,1\}^* \to \mathbb{N}$ be defined as follows. For every string $P \in \{0,1\}^*$, if P represents a NAND-TM program such that when P is executed on the input 0 then it halts within M steps then $T_{BB}(P) = M$. Otherwise (if P does not represent a NAND-TM program, or it is a program that does not halt on 0), $T_{BB}(P) = 0$. Prove that T_{BB} is uncomputable.

2. Let $TOWER(n)$ denote the number $\underbrace{2^{\cdot^{\cdot^{2}}}}_{n \text{ times}}$ (that is, a "tower of powers of two" of height n). To get a sense of how fast this function grows, $TOWER(1) = 2$, $TOWER(2) = 2^2 = 4$, $TOWER(3) = 2^{2^2} = 16$, $TOWER(4) = 2^{16} = 65536$ and $TOWER(5) = 2^{65536}$ which

is about 10^{20000}. $TOWER(6)$ is already a number that is too big to write even in scientific notation. Define $NBB : \mathbb{N} \to \mathbb{N}$ (for "NAND-TM Busy Beaver") to be the function $NBB(n) = \max_{P \in \{0,1\}^n} T_{BB}(P)$ where T_{BB} is as defined in Question 6.1. Prove that NBB grows *faster* than $TOWER$, in the sense that $TOWER(n) = o(NBB(n))$. See footnote for hint[9]

[9] You will not need to use very specific properties of the $TOWER$ function in this exercise. For example, $NBB(n)$ also grows faster than the Ackerman function.

■

9.7 BIBLIOGRAPHICAL NOTES

The cartoon of the Halting problem in Fig. 9.1 and taken from Charles Cooper's website, Copyright 2019 Charles F. Cooper.

Section 7.2 in [MM11] gives a highly recommended overview of uncomputability. Gödel, Escher, Bach [Hof99] is a classic popular science book that touches on uncomputability, and unprovability, and specifically Gödel's Theorem that we will see in Chapter 11. See also the recent book by Holt [Hol18].

The history of the definition of a function is intertwined with the development of mathematics as a field. For many years, a function was identified (as per Euler's quote above) with the means to calculate the output from the input. In the 1800's, with the invention of the Fourier series and with the systematic study of continuity and differentiability, people have started looking at more general kinds of functions, but the modern definition of a function as an arbitrary mapping was not yet universally accepted. For example, in 1899 Poincare wrote *"we have seen a mass of bizarre functions which appear to be forced to resemble as little as possible honest functions which serve some purpose. ... they are invented on purpose to show that our ancestor's reasoning was at fault, and we shall never get anything more than that out of them"*. Some of this fascinating history is discussed in [Gra83; Kle91; Lüt02; Gra05].

The existence of a universal Turing machine, and the uncomputability of *HALT* was first shown by Turing in his seminal paper [Tur37], though closely related results were shown by Church a year before. These works built on Gödel's 1931 *incompleteness theorem* that we will discuss in Chapter 11.

Some universal Turing machines with a small alphabet and number of states are given in [Rog96], including a single-tape universal Turing machine with the binary alphabet and with less than 25 states; see also the survey [WN09]. Adam Yedidia has written software to help in producing Turing machines with a small number of states. This is related to the recreational pastime of "Code Golfing" which is about solving a certain computational task using the as short as possible program. Finding "highly complex" small Turing machine is also

related to the "Busy Beaver" problem, see Exercise 9.13 and the survey [Aar20].

The diagonalization argument used to prove uncomputability of F^* is derived from Cantor's argument for the uncountability of the reals discussed in Chapter 2.

Christopher Strachey was an English computer scientist and the inventor of the CPL programming language. He was also an early artificial intelligence visionary, programming a computer to play Checkers and even write love letters in the early 1950's, see this New Yorker article and this website.

Rice's Theorem was proven in [Ric53]. It is typically stated in a form somewhat different than what we used, see Exercise 9.11.

We do not discuss in the chapter the concept of *recursively enumerable* languages, but it is covered briefly in Exercise 9.10. As usual, we use function, as opposed to language, notation.

Learning Objectives:

- See that Turing completeness is not always a good thing.
- Another example of an always-halting formalism: *context-free grammars* and *simply typed λ calculus*.
- The pumping lemma for non context-free functions.
- Examples of computable and uncomputable *semantic properties* of regular expressions and context-free grammars.

10
Restricted computational models

> *"Happy families are all alike; every unhappy family is unhappy in its own way"*, Leo Tolstoy (opening of the book "Anna Karenina").

We have seen that many models of computation are *Turing equivalent*, including Turing machines, NAND-TM/NAND-RAM programs, standard programming languages such as C/Python/Javascript, as well as other models such as the λ calculus and even the game of life. The flip side of this is that for all these models, Rice's theorem (Theorem 9.15) holds as well, which means that any semantic property of programs in such a model is *uncomputable*.

The uncomputability of halting and other semantic specification problems for Turing equivalent models motivates **restricted computational models** that are (**a**) powerful enough to capture a set of functions useful for certain applications but (**b**) weak enough that we can still solve semantic specification problems on them. In this chapter we discuss several such examples.

Big Idea 14 We can use *restricted computational models* to bypass limitations such as uncomputability of the Halting problem and Rice's Theorem. Such models can compute only a restricted subclass of functions, but allow to answer at least some *semantic questions* on programs.

10.1 TURING COMPLETENESS AS A BUG

We have seen that seemingly simple computational models or systems can turn out to be Turing complete. The following webpage lists several examples of formalisms that "accidentally" turned out to Turing complete, including supposedly limited languages such as the C preprocessor, CSS, (certain variants of) SQL, sendmail configuration, as well as games such as Minecraft, Super Mario, and the card game

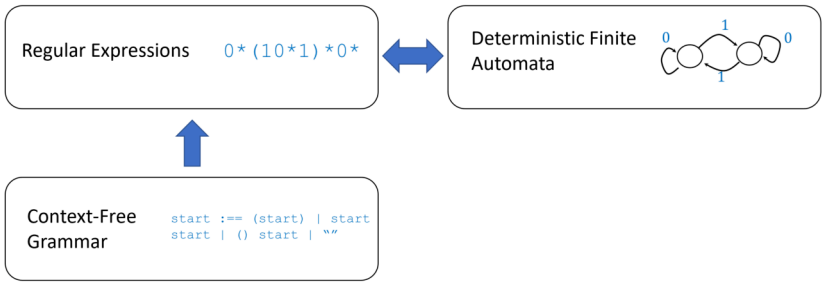


Figure 10.1: Some restricted computational models. We have already seen two equivalent restricted models of computation: regular expressions and deterministic finite automata. We show a more powerful model: context-free grammars. We also present tools to demonstrate that some functions *can not* be computed in these models.

"Magic: The Gathering". Turing completeness is not always a good thing, as it means that such formalisms can give rise to arbitrarily complex behavior. For example, the postscript format (a precursor of PDF) is a Turing-complete programming language meant to describe documents for printing. The expressive power of postscript can allow for short descriptions of very complex images, but it also gave rise to some nasty surprises, such as the attacks described in this page ranging from using infinite loops as a denial of service attack, to accessing the printer's file system.

■ **Example 10.1 — The DAO Hack.** An interesting recent example of the pitfalls of Turing-completeness arose in the context of the cryptocurrency Ethereum. The distinguishing feature of this currency is the ability to design "smart contracts" using an expressive (and in particular Turing-complete) programming language. In our current "human operated" economy, Alice and Bob might sign a contract to agree that if condition X happens then they will jointly invest in Charlie's company. Ethereum allows Alice and Bob to create a joint venture where Alice and Bob pool their funds together into an account that will be governed by some program P that decides under what conditions it disburses funds from it. For example, one could imagine a piece of code that interacts between Alice, Bob, and some program running on Bob's car that allows Alice to rent out Bob's car without any human intervention or overhead.

Specifically Ethereum uses the Turing-complete programming language solidity which has a syntax similar to JavaScript. The flagship of Ethereum was an experiment known as The "Decentralized Autonomous Organization" or The DAO. The idea was to create a smart contract that would create an autonomously run decentralized venture capital fund, without human managers, where shareholders could decide on investment opportunities. The

DAO was at the time the biggest crowdfunding success in history. At its height the DAO was worth 150 million dollars, which was more than ten percent of the total Ethereum market. Investing in the DAO (or entering any other "smart contract") amounts to providing your funds to be run by a computer program. i.e., "code is law", or to use the words the DAO described itself: *"The DAO is borne from immutable, unstoppable, and irrefutable computer code"*. Unfortunately, it turns out that (as we saw in Chapter 9) understanding the behavior of computer programs is quite a hard thing to do. A hacker (or perhaps, some would say, a savvy investor) was able to fashion an input that caused the DAO code to enter into an infinite recursive loop in which it continuously transferred funds into the hacker's account, thereby cleaning out about 60 million dollars out of the DAO. While this transaction was "legal" in the sense that it complied with the code of the smart contract, it was obviously not what the humans who wrote this code had in mind. The Ethereum community struggled with the response to this attack. Some tried the "Robin Hood" approach of using the same loophole to drain the DAO funds into a secure account, but it only had limited success. Eventually, the Ethereum community decided that the code can be mutable, stoppable, and refutable. Specifically, the Ethereum maintainers and miners agreed on a "hard fork" (also known as a "bailout") to revert history to before the hacker's transaction occurred. Some community members strongly opposed this decision, and so an alternative currency called Ethereum Classic was created that preserved the original history.

10.2 CONTEXT FREE GRAMMARS

If you have ever written a program, you've experienced a *syntax error*. You probably also had the experience of your program entering into an *infinite loop*. What is less likely is that the compiler or interpreter entered an infinite loop while trying to figure out if your program has a syntax error.

When a person designs a programming language, they need to determine its *syntax*. That is, the designer decides which strings corresponds to valid programs, and which ones do not (i.e., which strings contain a syntax error). To ensure that a compiler or interpreter always halts when checking for syntax errors, language designers typically *do not* use a general Turing-complete mechanism to express their syntax. Rather they use a *restricted* computational model. One of the most popular choices for such models is *context free grammars*.

To explain context free grammars, let us begin with a canonical example. Consider the function $ARITH : \Sigma^* \to \{0,1\}$ that takes as input a string x over the alphabet $\Sigma = \{(,),+,-,\times,\div,0,1,2,3,4,5,6,7,8,9\}$ and returns 1 if and only if the string x represents a valid arithmetic expression. Intuitively, we build expressions by applying an operation such as $+$,$-$,$\times$ or $\div$ to smaller expressions, or enclosing them in parentheses, where the "base case" corresponds to expressions that are simply numbers. More precisely, we can make the following definitions:

- A *digit* is one of the symbols $0,1,2,3,4,5,6,7,8,9$.
- A *number* is a sequence of digits. (For simplicity we drop the condition that the sequence does not have a leading zero, though it is not hard to encode it in a context-free grammar as well.)
- An *operation* is one of $+,-,\times,\div$
- An *expression* has either the form "*number*", the form "*sub-expression1 operation sub-expression2*", or the form "(*sub-expression1*)", where "sub-expression1" and "sub-expression2" are themselves expressions. (Note that this is a *recursive* definition.)

A context free grammar (CFG) is a formal way of specifying such conditions. A CFG consists of a set of *rules* that tell us how to generate strings from smaller components. In the above example, one of the rules is "if $exp1$ and $exp2$ are valid expressions, then $exp1 \times exp2$ is also a valid expression"; we can also write this rule using the shorthand $expression \Rightarrow expression \times expression$. As in the above example, the rules of a context-free grammar are often *recursive*: the rule $expression \Rightarrow expression \times expression$ defines valid expressions in terms of itself. We now formally define context-free grammars:

Definition 10.2 — Context Free Grammar. Let Σ be some finite set. A *context free grammar* (*CFG*) *over* Σ is a triple (V,R,s) such that:

- V, known as the *variables*, is a set disjoint from Σ.
- $s \in V$ is known as the *initial variable*.
- R is a set of *rules*. Each rule is a pair (v,z) with $v \in V$ and $z \in (\Sigma \cup V)^*$. We often write the rule (v,z) as $v \Rightarrow z$ and say that the string z *can be derived* from the variable v.

■ **Example 10.3 — Context free grammar for arithmetic expressions.** The example above of well-formed arithmetic expressions can be captured formally by the following context free grammar:

- The alphabet Σ is $\{(,),+,-,\times,\div,0,1,2,3,4,5,6,7,8,9\}$
- The variables are $V = \{expression\ ,\ number\ ,\ digit\ ,\ operation\}$.
- The rules are the set R containing the following 19 rules:
  - The 4 rules $operation \Rightarrow +$, $operation \Rightarrow -$, $operation \Rightarrow \times$, and $operation \Rightarrow \div$.
  - The 10 rules $digit \Rightarrow 0$,..., $digit \Rightarrow 9$.
  - The rule $number \Rightarrow digit$.
  - The rule $number \Rightarrow digit\ number$.
  - The rule $expression \Rightarrow number$.
  - The rule $expression \Rightarrow expression\ operation\ expression$.
  - The rule $expression \Rightarrow (expression)$.
- The starting variable is $expression$

People use many different notations to write context free grammars. One of the most common notations is the Backus–Naur form. In this notation we write a rule of the form $v \Rightarrow a$ (where v is a variable and a is a string) in the form `<v> := a`. If we have several rules of the form $v \mapsto a$, $v \mapsto b$, and $v \mapsto c$ then we can combine them as `<v> := a|b|c`. (In words we say that v can derive either a, b, or c.) For example, the Backus-Naur description for the context free grammar of Example 10.3 is the following (using ASCII equivalents for operations):

```
operation  := +|-|*|/
digit      := 0|1|2|3|4|5|6|7|8|9
number     := digit|digit number
expression := number|expression operation
 ↪  expression|(expression)
```

Another example of a context free grammar is the "matching parentheses" grammar, which can be represented in Backus-Naur as follows:

```
match  := ""|match match|(match)
```

A string over the alphabet { `(`,`)` } can be generated from this grammar (where `match` is the starting expression and `""` corresponds to the empty string) if and only if it consists of a matching set of parentheses.

In contrast, by Lemma 6.20 there is no regular expression that matches a string x if and only if x contains a valid sequence of matching parentheses.

10.2.1 Context-free grammars as a computational model

We can think of a context-free grammar over the alphabet Σ as defining a function that maps every string x in Σ^* to 1 or 0 depending on whether x can be generated by the rules of the grammars. We now make this definition formally.

Definition 10.4 — Deriving a string from a grammar. If $G = (V, R, s)$ is a context-free grammar over Σ, then for two strings $\alpha, \beta \in (\Sigma \cup V)^*$ we say that β *can be derived in one step* from α, denoted by $\alpha \Rightarrow_G \beta$, if we can obtain β from α by applying one of the rules of G. That is, we obtain β by replacing in α one occurrence of the variable v with the string z, where $v \Rightarrow z$ is a rule of G.

We say that β *can be derived* from α, denoted by $\alpha \Rightarrow^*_G \beta$, if it can be derived by some finite number k of steps. That is, if there are $\alpha_1, \ldots, \alpha_{k-1} \in (\Sigma \cup V)^*$, so that $\alpha \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \cdots \Rightarrow_G \alpha_{k-1} \Rightarrow_G \beta$.

We say that $x \in \Sigma^*$ is *matched* by $G = (V, R, s)$ if x can be derived from the starting variable s (i.e., if $s \Rightarrow^*_G x$). We define the *function computed by* (V, R, s) to be the map $\Phi_{V,R,s} : \Sigma^* \to \{0,1\}$ such that $\Phi_{V,R,s}(x) = 1$ iff x is matched by (V, R, s). A function $F : \Sigma^* \to \{0,1\}$ is *context free* if $F = \Phi_{V,R,s}$ for some CFG (V, R, s). [1]

[1] As in the case of Definition 6.7 we can also use *language* rather than *function* notation and say that a language $L \subseteq \Sigma^*$ is *context free* if the function F such that $F(x) = 1$ iff $x \in L$ is context free.

A priori it might not be clear that the map $\Phi_{V,R,s}$ is computable, but it turns out that this is the case.

Theorem 10.5 — Context-free grammars always halt. For every CFG (V, R, s) over $\{0,1\}$, the function $\Phi_{V,R,s} : \{0,1\}^* \to \{0,1\}$ is computable.

As usual we restrict attention to grammars over $\{0,1\}$ although the proof extends to any finite alphabet Σ.

Proof. We only sketch the proof. We start with the observation we can convert every CFG to an equivalent version of *Chomsky normal form*, where all rules either have the form $u \to vw$ for variables u, v, w or the form $u \to \sigma$ for a variable u and symbol $\sigma \in \Sigma$, plus potentially the rule $s \to$ "" where s is the starting variable.

The idea behind such a transformation is to simply add new variables as needed, and so for example we can translate a rule such as $v \to u\sigma w$ into the three rules $v \to ur$, $r \to tw$ and $t \to \sigma$.

Using the Chomsky Normal form we get a natural recursive algorithm for computing whether $s \Rightarrow_G^* x$ for a given grammar G and string x. We simply try all possible guesses for the first rule $s \to uv$ that is used in such a derivation, and then all possible ways to partition x as a concatenation $x = x'x''$. If we guessed the rule and the partition correctly, then this reduces our task to checking whether $u \Rightarrow_G^* x'$ and $v \Rightarrow_G^* x''$, which (as it involves shorter strings) can be done recursively. The base cases are when x is empty or a single symbol, and can be easily handled.

■

R

Remark 10.6 — Parse trees. While we focus on the task of *deciding* whether a CFG matches a string, the algorithm to compute $\Phi_{V,R,s}$ actually gives more information than that. That is, on input a string x, if $\Phi_{V,R,s}(x) = 1$ then the algorithm yields the sequence of rules that one can apply from the starting vertex s to obtain the final string x. We can think of these rules as determining a *tree* with s being the *root* vertex and the sinks (or *leaves*) corresponding to the substrings of x that are obtained by the rules that do not have a variable in their second element. This tree is known as the *parse tree* of x, and often yields very useful information about the structure of x.

Often the first step in a compiler or interpreter for a programming language is a *parser* that transforms the source into the parse tree (also known as the abstract syntax tree). There are also tools that can automatically convert a description of a context-free grammars into a parser algorithm that computes the parse tree of a given string. (Indeed, the above recursive algorithm can be used to achieve this, but there are much more efficient versions, especially for grammars that have particular forms, and programming language designers often try to ensure their languages have these more efficient grammars.)

10.2.2 The power of context free grammars

Context free grammars can capture every regular expression:

Theorem 10.7 — Context free grammars and regular expressions. Let e be a regular expression over $\{0,1\}$, then there is a CFG (V, R, s) over $\{0,1\}$ such that $\Phi_{V,R,s} = \Phi_e$.

Proof. We prove the theorem by induction on the length of e. If e is an expression of one bit length, then $e = 0$ or $e = 1$, in which case we leave it to the reader to verify that there is a (trivial) CFG that

computes it. Otherwise, we fall into one of the following case: **case 1:** $e = e'e''$, **case 2:** $e = e'|e''$ or **case 3:** $e = (e')^*$ where in all cases e', e'' are shorter regular expressions. By the induction hypothesis, we can define grammars (V', R', s') and (V'', R'', s'') that compute $\Phi_{e'}$ and $\Phi_{e''}$ respectively. By renaming variables, we can also assume without loss of generality that V' and V'' are disjoint.

In case 1, we can define the new grammar as follows: we add a new starting variable $s \notin V \cup V'$ and the rule $s \mapsto s's''$. In case 2, we can define the new grammar as follows: we add a new starting variable $s \notin V \cup V'$ and the rules $s \mapsto s'$ and $s \mapsto s''$. Case 3 will be the only one that uses *recursion*. As before we add a new starting variable $s \notin V \cup V'$, but now add the rules $s \mapsto$ "" (i.e., the empty string) and also add, for every rule of the form $(s', \alpha) \in R'$, the rule $s \mapsto s\alpha$ to R.

We leave it to the reader as (a very good!) exercise to verify that in all three cases the grammars we produce capture the same function as the original expression.

■

It turns out that CFG's are strictly more powerful than regular expressions. In particular, as we've seen, the "matching parentheses" function *MATCHPAREN* can be computed by a context free grammar, whereas, as shown in Lemma 6.20, it cannot be computed by regular expressions. Here is another example:

Solved Exercise 10.1 — Context free grammar for palindromes. Let $PAL : \{0,1,;\}^* \to \{0,1\}$ be the function defined in Solved Exercise 6.4 where $PAL(w) = 1$ iff w has the form $u;u^R$. Then PAL can be computed by a context-free grammar

■

Solution:

A simple grammar computing PAL can be described using Backus–Naur notation:

```
start      := ; | 0 start 0 | 1 start 1
```

One can prove by induction that this grammar generates exactly the strings w such that $PAL(w) = 1$.

■

A more interesting example is computing the strings of the form $u;v$ that are *not* palindromes:

Solved Exercise 10.2 — Non-palindromes. Prove that there is a context free grammar that computes $NPAL : \{0,1,;\}^* \to \{0,1\}$ where $NPAL(w) = 1$ if $w = u;v$ but $v \neq u^R$.

■

Solution:

Using Backus–Naur notation we can describe such a grammar as follows

```
palindrome      := ; | 0 palindrome 0 | 1 palindrome 1
different       := 0 palindrome 1 | 1 palindrome 0
start           := different | 0 start | 1 start | start
 ↪  0 | start 1
```

In words, this means that we can characterize a string w such that $NPAL(w) = 1$ as having the following form

$$w = \alpha b u; u^R b' \beta$$

where α, β, u are arbitrary strings and $b \neq b'$. Hence we can generate such a string by first generating a palindrome $u; u^R$ (`palindrome` variable), then adding 0 on either the left or right and 1 on the opposite side to get something that is *not* a palindrome (`different` variable), and then we can add arbitrary number of 0's and 1's on either end (the `start` variable).

■

10.2.3 Limitations of context-free grammars (optional)

Even though context-free grammars are more powerful than regular expressions, there are some simple languages that are *not* captured by context free grammars. One tool to show this is the context-free grammar analog of the "pumping lemma" (Theorem 6.21):

Theorem 10.8 — Context-free pumping lemma. Let (V, R, s) be a CFG over Σ, then there is some numbers $n_0, n_1 \in \mathbb{N}$ such that for every $x \in \Sigma^*$ with $|x| > n_0$, if $\Phi_{V,R,s}(x) = 1$ then $x = abcde$ such that $|b| + |c| + |d| \leq n_1$, $|b| + |d| \geq 1$, and $\Phi_{V,R,s}(ab^k cd^k e) = 1$ for every $k \in \mathbb{N}$.

The context-free pumping lemma is even more cumbersome to state than its regular analog, but you can remember it as saying the following: *"If a long enough string is matched by a grammar, there must be a variable that is repeated in the derivation."*

Proof of Theorem 10.8. We only sketch the proof. The idea is that if the total number of symbols in the rules of the grammar is n_0, then the only way to get $|x| > n_0$ with $\Phi_{V,R,s}(x) = 1$ is to use *recursion*. That is, there must be some variable $v \in V$ such that we are able to

derive from v the value bvd for some strings $b, d \in \Sigma^*$, and then further on derive from v some string $c \in \Sigma^*$ such that bcd is a substring of x (in other words, $x = abcde$ for some $a, e \in \{0,1\}^*$). If we take the variable v satisfying this requirement with a minimum number of derivation steps, then we can ensure that $|bcd|$ is at most some constant depending on n_0 and we can set n_1 to be that constant ($n_1 = 10 \cdot |R| \cdot n_0$ will do, since we will not need more than $|R|$ applications of rules, and each such application can grow the string by at most n_0 symbols).

Thus by the definition of the grammar, we can repeat the derivation to replace the substring bcd in x with $b^k c d^k$ for every $k \in \mathbb{N}$ while retaining the property that the output of $\Phi_{V,R,s}$ is still one. Since bcd is a substring of x, we can write $x = abcde$ and are guaranteed that $ab^k c d^k e$ is matched by the grammar for every k.

■

Using Theorem 10.8 one can show that even the simple function $F : \{0,1\}^* \to \{0,1\}$ defined as follows:

$$F(x) = \begin{cases} 1 & x = ww \text{ for some } w \in \{0,1\}^* \\ 0 & \text{otherwise} \end{cases}$$

is not context free. (In contrast, the function $G : \{0,1\}^* \to \{0,1\}$ defined as $G(x) = 1$ iff $x = w_0 w_1 \cdots w_{n-1} w_{n-1} w_{n-2} \cdots w_0$ for some $w \in \{0,1\}^*$ and $n = |w|$ is context free, can you see why?.)

Solved Exercise 10.3 — Equality is not context-free. Let $EQ : \{0,1,;\}^* \to \{0,1\}$ be the function such that $EQ(x) = 1$ if and only if $x = u;u$ for some $u \in \{0,1\}^*$. Then EQ is not context free.

■

Solution:

We use the context-free pumping lemma. Suppose towards the sake of contradiction that there is a grammar G that computes EQ, and let n_0 be the constant obtained from Theorem 10.8.

Consider the string $x = 1^{n_0}0^{n_0};1^{n_0}0^{n_0}$, and write it as $x = abcde$ as per Theorem 10.8, with $|bcd| \leq n_0$ and with $|b| + |d| \geq 1$. By Theorem 10.8, it should hold that $EQ(ace) = 1$. However, by case analysis this can be shown to be a contradiction.

Firstly, unless b is on the left side of the ; separator and d is on the right side, dropping b and d will definitely make the two parts different. But if it is the case that b is on the left side and d is on the right side, then by the condition that $|bcd| \leq n_0$ we know that b is a string of only zeros and d is a string of only ones. If we drop b and d then since one of them is non-empty, we get that there are either

less zeroes on the left side than on the right side, or there are less ones on the right side than on the left side. In either case, we get that $EQ(ace) = 0$, obtaining the desired contradiction.

■

10.3 SEMANTIC PROPERTIES OF CONTEXT FREE LANGUAGES

As in the case of regular expressions, the limitations of context free grammars do provide some advantages. For example, emptiness of context free grammars is decidable:

Theorem 10.9 — Emptiness for CFG's is decidable. There is an algorithm that on input a context-free grammar G, outputs 1 if and only if Φ_G is the constant zero function.

Proof Idea:

The proof is easier to see if we transform the grammar to Chomsky Normal Form as in Theorem 10.5. Given a grammar G, we can recursively define a non-terminal variable v to be *non-empty* if there is either a rule of the form $v \Rightarrow \sigma$, or there is a rule of the form $v \Rightarrow uw$ where both u and w are non-empty. Then the grammar is non-empty if and only if the starting variable s is non-empty.

⋆

Proof of Theorem 10.9. We assume that the grammar G in Chomsky Normal Form as in Theorem 10.5. We consider the following procedure for marking variables as "non-empty":

1. We start by marking all variables v that are involved in a rule of the form $v \Rightarrow \sigma$ as non-empty.

2. We then continue to mark v as non-empty if it is involved in a rule of the form $v \Rightarrow uw$ where u, w have been marked before.

We continue this way until we cannot mark any more variables. We then declare that the grammar is empty if and only if s has not been marked. To see why this is a valid algorithm, note that if a variable v has been marked as "non-empty" then there is some string $\alpha \in \Sigma^*$ that can be derived from v. On the other hand, if v has not been marked, then every sequence of derivations from v will always have a variable that has not been replaced by alphabet symbols. Hence in particular Φ_G is the all zero function if and only if the starting variable s is not marked "non-empty".

■

10.3.1 Uncomputability of context-free grammar equivalence (optional)

By analogy to regular expressions, one might have hoped to get an algorithm for deciding whether two given context free grammars are equivalent. Alas, no such luck. It turns out that the equivalence problem for context free grammars is *uncomputable*. This is a direct corollary of the following theorem:

Theorem 10.10 — Fullness of CFG's is uncomputable. For every set Σ, let $CFGFULL_\Sigma$ be the function that on input a context-free grammar G over Σ, outputs 1 if and only if G computes the constant 1 function. Then there is some finite Σ such that $CFGFULL_\Sigma$ is uncomputable.

Theorem 10.10 immediately implies that equivalence for context-free grammars is uncomputable, since computing "fullness" of a grammar G over some alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_{k-1}\}$ corresponds to checking whether G is equivalent to the grammar $s \Rightarrow$ ""$|s\sigma_0| \cdots |s\sigma_{k-1}$. Note that Theorem 10.10 and Theorem 10.9 together imply that context-free grammars, unlike regular expressions, are *not* closed under complement. (Can you see why?) Since we can encode every element of Σ using $\lceil \log |\Sigma| \rceil$ bits (and this finite encoding can be easily carried out within a grammar) Theorem 10.10 implies that fullness is also uncomputable for grammars over the binary alphabet.

Proof Idea:

We prove the theorem by reducing from the Halting problem. To do that we use the notion of *configurations* of NAND-TM programs, as defined in Definition 8.8. Recall that a *configuration* of a program P is a binary string s that encodes all the information about the program in the current iteration.

We define Σ to be $\{0,1\}$ plus some separator characters and define $INVALID_P : \Sigma^* \to \{0,1\}$ to be the function that maps every string $L \in \Sigma^*$ to 1 if and only if L does *not* encode a sequence of configurations that correspond to a valid halting history of the computation of P on the empty input.

The heart of the proof is to show that $INVALID_P$ is context-free. Once we do that, we see that P halts on the empty input if and only if $INVALID_P(L) = 1$ for *every* L. To show that, we will encode the list in a special way that makes it amenable to deciding via a context-free grammar. Specifically we will reverse all the odd-numbered strings.

⋆

Proof of Theorem 10.10. We only sketch the proof. We will show that if we can compute *CFGFULL* then we can solve *HALTONZERO*, which has been proven uncomputable in Theorem 9.9. Let M be an input

Turing machine for *HALTONZERO*. We will use the notion of *configurations* of a Turing machine, as defined in Definition 8.8.

Recall that a *configuration* of Turing machine M and input x captures the full state of M at some point of the computation. The particular details of configurations are not so important, but what you need to remember is that:

- A configuration can be encoded by a binary string $\sigma \in \{0,1\}^*$.
- The *initial* configuration of M on the input 0 is some fixed string.
- A *halting configuration* will have the value a certain state (which can be easily "read off" from it) set to 1.
- If σ is a configuration at some step i of the computation, we denote by $NEXT_M(\sigma)$ as the configuration at the next step. $NEXT_M(\sigma)$ is a string that agrees with σ on all but a constant number of coordinates (those encoding the position corresponding to the head position and the two adjacent ones). On those coordinates, the value of $NEXT_M(\sigma)$ can be computed by some finite function.

We will let the alphabet $\Sigma = \{0,1\} \cup \{\|, \#\}$. A *computation history* of M on the input 0 is a string $L \in \Sigma$ that corresponds to a list $\|\sigma_0\#\sigma_1\|\sigma_2\#\sigma_3 \cdots \sigma_{t-2}\|\sigma_{t-1}\#$ (i.e., $\|$ comes before an even numbered block, and $\#$ comes before an odd numbered one) such that if i is even then σ_i is the string encoding the configuration of P on input 0 at the beginning of its i-th iteration, and if i is odd then it is the same except the string is *reversed*. (That is, for odd i, $rev(\sigma_i)$ encodes the configuration of P on input 0 at the beginning of its i-th iteration.) Reversing the odd-numbered blocks is a technical trick to ensure that the function $INVALID_M$ we define below is context free.

We now define $INVALID_M : \Sigma^* \to \{0,1\}$ as follows:

$$INVALID_M(L) = \begin{cases} 0 & L \text{ is a valid computation history of } M \text{ on } 0 \\ 1 & \text{otherwise} \end{cases}$$

We will show the following claim:

CLAIM: $INVALID_M$ is context-free.

The claim implies the theorem. Since M halts on 0 if and only if there exists a valid computation history, $INVALID_M$ is the constant one function if and only if M does *not* halt on 0. In particular, this allows us to reduce determining whether M halts on 0 to determining whether the grammar G_M corresponding to $INVALID_M$ is full.

We now turn to the proof of the claim. We will not show all the details, but the main point $INVALID_M(L) = 1$ if *at least one* of the following three conditions hold:

1. L is not of the right format, i.e. not of the form $\langle$binary-string$\rangle\#\langle$binary-string$\rangle\|\langle$binary-string$\rangle\#\cdots$.

2. L contains a substring of the form $\|\sigma\#\sigma'\|$ such that $\sigma' \neq rev(NEXT_P(\sigma))$

3. L contains a substring of the form $\#\sigma\|\sigma'\#$ such that $\sigma' \neq NEXT_P(rev(\sigma))$

Since context-free functions are closed under the OR operation, the claim will follow if we show that we can verify conditions 1, 2 and 3 via a context-free grammar.

For condition 1 this is very simple: checking that L *is* of the correct format can be done using a regular expression. Since regular expressions are closed under negation, this means that checking that L is *not* of this format can also be done by a regular expression and hence by a context-free grammar.

For conditions 2 and 3, this follows via very similar reasoning to that showing that the function F such that $F(u\#v) = 1$ iff $u \neq rev(v)$ is context-free, see Solved Exercise 10.2. After all, the $NEXT_M$ function only modifies its input in a constant number of places. We leave filling out the details as an exercise to the reader. Since $INVALID_M(L) = 1$ if and only if L satisfies one of the conditions 1., 2. or 3., and all three conditions can be tested for via a context-free grammar, this completes the proof of the claim and hence the theorem.

■

10.4 SUMMARY OF SEMANTIC PROPERTIES FOR REGULAR EXPRESSIONS AND CONTEXT-FREE GRAMMARS

To summarize, we can often trade *expressiveness* of the model for *amenability to analysis*. If we consider computational models that are *not* Turing complete, then we are sometimes able to bypass Rice's Theorem and answer certain semantic questions about programs in such models. Here is a summary of some of what is known about semantic questions for the different models we have seen.

Table 10.1: Computability of semantic properties

Model	**Halting**	**Emptiness**	**Equivalence**
Regular expressions	Computable	Computable	Computable
Context free grammars	Computable	Computable	Uncomputable
Turing-complete models	Uncomputable	Uncomputable	Uncomputable

Chapter Recap

- The uncomputability of the Halting problem for general models motivates the definition of restricted computational models.
- In some restricted models we can answer *semantic* questions such as: does a given program terminate, or do two programs compute the same function?
- *Regular expressions* are a restricted model of computation that is often useful to capture tasks of string matching. We can test efficiently whether an expression matches a string, as well as answer questions such as Halting and Equivalence.
- *Context free grammars* is a stronger, yet still not Turing complete, model of computation. The halting problem for context free grammars is computable, but equivalence is not computable.

10.5 EXERCISES

Exercise 10.1 — Closure properties of context-free functions. Suppose that $F, G : \{0,1\}^* \to \{0,1\}$ are context free. For each one of the following definitions of the function H, either prove that H is always context free or give a counterexample for regular F, G that would make H not context free.

1. $H(x) = F(x) \vee G(x)$.
2. $H(x) = F(x) \wedge G(x)$
3. $H(x) = NAND(F(x), G(x))$.
4. $H(x) = F(x^R)$ where x^R is the reverse of x: $x^R = x_{n-1}x_{n-2}\cdots x_o$ for $n = |x|$.
5. $H(x) = \begin{cases} 1 & x = uv \text{ s.t. } F(u) = G(v) = 1 \\ 0 & \text{otherwise} \end{cases}$
6. $H(x) = \begin{cases} 1 & x = uu \text{ s.t. } F(u) = G(u) = 1 \\ 0 & \text{otherwise} \end{cases}$
7. $H(x) = \begin{cases} 1 & x = uu^R \text{ s.t. } F(u) = G(u) = 1 \\ 0 & \text{otherwise} \end{cases}$

■

Exercise 10.2 Prove that the function $F : \{0,1\}^* \to \{0,1\}$ such that $F(x) = 1$ if and only if $|x|$ is a power of two is not context free.

■

Exercise 10.3 — Syntax for programming languages. Consider the following syntax of a "programming language" whose source can be written using the ASCII character set:

- *Variables* are obtained by a sequence of letters, numbers and underscores, but can't start with a number.
- A *statement* has either the form `foo = bar;` where `foo` and `bar` are variables, or the form `IF (foo) BEGIN ... END` where `...` is list of one or more statements, potentially separated by newlines.

A *program* in our language is simply a sequence of statements (possibly separated by newlines or spaces).

1. Let $VAR : \{0,1\}^* \to \{0,1\}$ be the function that given a string $x \in \{0,1\}^*$, outputs 1 if and only if x corresponds to an ASCII encoding of a valid variable identifier. Prove that *VAR* is regular.
2. Let $SYN : \{0,1\}^* \to \{0,1\}$ be the function that given a string $s \in \{0,1\}^*$, outputs 1 if and only if s is an ASCII encoding of a valid program in our language. Prove that *SYN* is context free. (You do not have to specify the full formal grammar for *SYN*, but you need to show that such a grammar exists.)
3. Prove that *SYN* is not regular. See footnote for hint[2]

■

[2] Try to see if you can "embed" in some way a function that looks similar to *MATCHPAREN* in *SYN*, so you can use a similar proof. Of course for a function to be non-regular, it does not need to utilize literal parentheses symbols.

10.6 BIBLIOGRAPHICAL NOTES

As in the case of regular expressions, there are many resources available that cover context-free grammar in great detail. Chapter 2 of [Sip97] contains many examples of context-free grammars and their properties. There are also websites such as Grammophone where you can input grammars, and see what strings they generate, as well as some of the properties that they satisfy.

The adjective "context free" is used for CFG's because a rule of the form $v \mapsto a$ means that we can *always* replace v with the string a, no matter what is the *context* in which v appears. More generally, we might want to consider cases where the replacement rules depend on the context. This gives rise to the notion of *general* (*aka "Type 0"*) *grammars* that allow rules of the form $a \Rightarrow b$ where both a and b are strings over $(V \cup \Sigma)^*$. The idea is that if, for example, we wanted to enforce the condition that we only apply some rule such as $v \mapsto 0w1$ when v is surrounded by three zeroes on both sides, then we could do so by adding a rule of the form $000v000 \mapsto 0000w1000$ (and of course we can add much more general conditions). Alas, this generality

comes at a cost - general grammars are Turing complete and hence their halting problem is uncomputable. That is, there is no algorithm A that can determine for every general grammar G and a string x, whether or not the grammar G generates x.

The Chomsky Hierarchy is a hierarchy of grammars from the least restrictive (most powerful) Type 0 grammars, which correspond to *recursively enumerable* languages (see Exercise 9.10) to the most restrictive Type 3 grammars, which correspond to regular languages. Context-free languages correspond to Type 2 grammars. Type 1 grammars are *context sensitive grammars*. These are more powerful than context-free grammars but still less powerful than Turing machines. In particular functions/languages corresponding to context-sensitive grammars are always computable, and in fact can be computed by a linear bounded automatons which are non-deterministic algorithms that take $O(n)$ space. For this reason, the class of functions/languages corresponding to context-sensitive grammars is also known as the complexity class **NSPACE**$O(n)$; we discuss space-bounded complexity in Chapter 17). While Rice's Theorem implies that we cannot compute any non-trivial semantic property of Type 0 grammars, the situation is more complex for other types of grammars: some semantic properties can be determined and some cannot, depending on the grammar's place in the hierarchy.

Learning Objectives:

- More examples of uncomputable functions that are not as tied to computation.
- Gödel's incompleteness theorem - a result that shook the world of mathematics in the early 20th century.

11
Is every theorem provable?

"Take any definite unsolved problem, such as ... the existence of an infinite number of prime numbers of the form $2^n + 1$. However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, the firm conviction that their solution must follow by a finite number of purely logical processes..."
"...This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus.", David Hilbert, 1900.

"The meaning of a statement is its method of verification.", Moritz Schlick, 1938 (aka "The verification principle" of logical positivism)

The problems shown uncomputable in Chapter 9, while natural and important, still intimately involved NAND-TM programs or other computing mechanisms in their definitions. One could perhaps hope that as long as we steer clear of functions whose inputs are themselves programs, we can avoid the "curse of uncomputability". Alas, we have no such luck.

In this chapter we will see an example of a natural and seemingly "computation free" problem that nevertheless turns out to be uncomputable: solving Diophantine equations. As a corollary, we will see one of the most striking results of 20th century mathematics: *Gödel's Incompleteness Theorem*, which showed that there are some mathematical statements (in fact, in number theory) that are *inherently unprovable*. We will actually start with the latter result, and then show the former.

This chapter: A non-mathy overview
The marquee result of this chapter is Gödel's Incompleteness Theorem, which states that for every proof system, there are some statements about arithmetic that are true but *unprovable* in this system. But more than that we will see a deep connec-

tion between *uncomputability* and *unprovability*. For example, the uncomputability of the Halting problem immediately gives rise to the existence of unprovable statements about Turing machines. To even state Gödel's Incompleteness Theorem we will need to formally define the notion of a "proof system". We give a very general definition, that encompasses all types of "axioms + inference rules" systems used in logic and math. We will then build up the machinery to encode computation using arithmetic that will enable us to prove Gödel's Theorem.

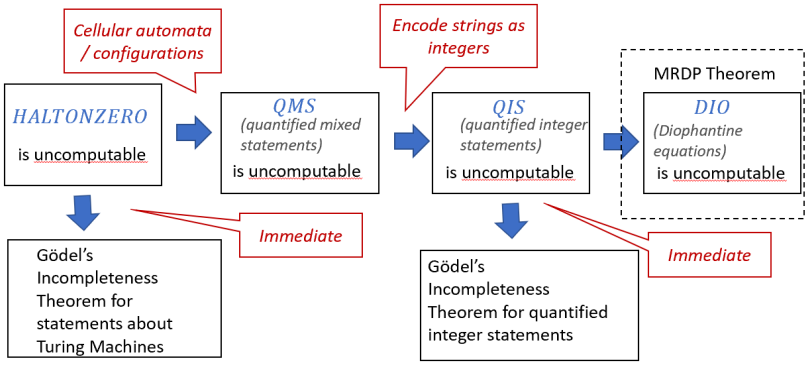


Figure 11.1: Outline of the results of this chapter. One version of Gödel's Incompleteness Theorem is an immediate consequence of the uncomputability of the Halting problem. To obtain the theorem as originally stated (for statements about the integers) we first prove that the *QMS* problem of determining truth of quantified statements involving both integers and strings is uncomputable. We do so using the notion of *Turing Machine configurations* but there are alternative approaches to do so as well, see Remark 11.14.

11.1 HILBERT'S PROGRAM AND GÖDEL'S INCOMPLETENESS THEOREM

"And what are these ...vanishing increments? They are neither finite quantities, nor quantities infinitely small, nor yet nothing. May we not call them the ghosts of departed quantities?", George Berkeley, Bishop of Cloyne, 1734.

The 1700's and 1800's were a time of great discoveries in mathematics but also of several crises. The discovery of calculus by Newton and Leibnitz in the late 1600's ushered a golden age of problem solving. Many longstanding challenges succumbed to the new tools that were discovered, and mathematicians got ever better at doing some truly impressive calculations. However, the rigorous foundations behind these calculations left much to be desired. Mathematicians manipulated infinitesimal quantities and infinite series cavalierly, and while most of the time they ended up with the correct results, there were a few strange examples (such as trying to calculate the value of the infinite series $1 - 1 + 1 - 1 + 1 + ...$) which seemed to give out different answers depending on the method of calculation. This led to a growing sense of unease in the foundations of the subject

which was addressed in the works of mathematicians such as Cauchy, Weierstrass, and Riemann, who eventually placed analysis on firmer foundations, giving rise to the ϵ's and δ's that students taking honors calculus grapple with to this day.

In the beginning of the 20th century, there was an effort to replicate this effort, in greater rigor, to all parts of mathematics. The hope was to show that all the true results of mathematics can be obtained by starting with a number of axioms, and deriving theorems from them using logical rules of inference. This effort was known as the *Hilbert program*, named after the influential mathematician David Hilbert.

Alas, it turns out the results we've seen dealt a devastating blow to this program, as was shown by Kurt Gödel in 1931:

> **Theorem 11.1 — Gödel's Incompleteness Theorem: informal version.** For every sound proof system V for sufficiently rich mathematical statements, there is a mathematical statement that is *true* but is not *provable* in V.

11.1.1 Defining "Proof Systems"

Before proving Theorem 11.1, we need to define "proof systems" and even formally define the notion of a "mathematical statement". In geometry and other areas of mathematics, proof systems are often defined by starting with some basic assumptions or *axioms* and then deriving more statements by using *inference rules* such as the famous Modus Ponens, but what axioms shall we use? What rules? We will use an extremely general notion of proof systems, not even restricting ourselves to ones that have the form of axioms and inference.

Mathematical statements. At the highest level, a mathematical statement is simply a piece of text, which we can think of as a *string* $x \in \{0,1\}^*$. Mathematical statements contain assertions whose truth does not depend on any empirical fact, but rather only on properties of abstract objects. For example, the following is a mathematical statement:[1]

[1] This happens to be a *false* statement.

> *"The number* 2,696,635,869,504,783,333,238,805,675,613, 588,278,597,832,162,617,892,474,670,798,113 *is prime".*

Mathematical statements do not have to involve numbers. They can assert properties of any other mathematical object including sets, strings, functions, graphs and yes, even *programs*. Thus, another example of a mathematical statement is the following:[2]

[2] It is unknown whether this statement is true or false.

> The following Python function halts on every positive integer `n`

```
def f(n):
    if n==1: return 1
    return f(3*n+1) if n % 2 else f(n//2)
```

Proof systems. A *proof* for a statement $x \in \{0,1\}^*$ is another piece of text $w \in \{0,1\}^*$ that certifies the truth of the statement asserted in x. The conditions for a valid proof system are:

1. (*Effectiveness*) Given a statement x and a proof w, there is an algorithm to verify whether or not w is a valid proof for x. (For example, by going line by line and checking that each line follows from the preceding ones using one of the allowed inference rules.)

2. (*Soundness*) If there is a valid proof w for x then x is true.

These are quite minimal requirements for a proof system. Requirement 2 (soundness) is the very definition of a proof system: you shouldn't be able to prove things that are not true. Requirement 1 is also essential. If there is no set of rules (i.e., an algorithm) to check that a proof is valid then in what sense is it a proof system? We could replace it with a system where the "proof" for a statement x is "trust me: it's true".

We formally define proof systems as an algorithm V where $V(x,w) = 1$ holds if the string w is a valid proof for the statement x. Even if x is true, the string w does not have to be a valid proof for it (there are plenty of wrong proofs for true statements such as `4=2+2`) but if w is a valid proof for x then x must be true.

Definition 11.2 — Proof systems. Let $\mathcal{T} \subseteq \{0,1\}^*$ be some set (which we consider the "true" statements). A *proof system* for $\mathcal{T}$ is an algorithm V that satisfies:

1. (*Effectiveness*) For every $x, w \in \{0,1\}^*$, $V(x,w)$ halts with an output of either 0 or 1.

2. (*Soundness*) For every $x \notin \mathcal{T}$ and $w \in \{0,1\}^*$, $V(x,w) = 0$.

A true statement $x \in \mathcal{T}$ is *unprovable* (with respect to V) if for every $w \in \{0,1\}^*$, $V(x,w) = 0$. We say that V is *complete* if there does not exist a true statement x that is unprovable with respect to V.

Big Idea 15 A *proof* is just a string of text whose meaning is given by a *verification algorithm*.

11.2 GÖDEL'S INCOMPLETENESS THEOREM: COMPUTATIONAL VARIANT

Our first formalization of Theorem 11.1 involves statements about Turing machines. We let $\mathcal{H}$ be the set of strings $x \in \{0,1\}^*$ that have the form "Turing machine M halts on the zero input".

Theorem 11.3 — Gödel's Incompleteness Theorem: computational variant. There does not exist a complete proof system for $\mathcal{H}$.

Proof Idea:

If we had such a complete and sound proof system then we could solve the *HALTONZERO* problem. On input a Turing machine M, we would search all purported proofs w and halt as soon as we find a proof of either "M halts on zero" or "M does not halt on zero". If the system is sound and complete then we will eventually find such a proof, and it will provide us with the correct output.

⋆

Proof of Theorem 11.3. Assume for the sake of contradiction that there was such a proof system V. We will use V to build an algorithm A that computes *HALTONZERO*, hence contradicting Theorem 9.9. Our algorithm A will work as follows:

Algorithm 11.4 — Halting from proofs.

Input: Turing machine M
Output: 1 M if halts on the
Input: 0; 0 otherwise.

```
for n = 1, 2, 3, ... do
    for w ∈ {0,1}^n do
        if V("M halts on 0", w) = 1 then
            return 1
        end if
        if V("M does not halt on 0", w) = 1 then
            return 0
        end if
    end for
end for
```

If M halts on 0 then under our assumption there exists w that proves this fact, and so when Algorithm A reaches $n = |w|$ we will eventually find this w and output 1, unless we already halted before. But we cannot halt before and output a wrong answer because it would contradict the soundness of the proof system. Similarly, this shows that if M does *not* halt on 0 then (since we assume there is a

proof of this fact too) our algorithm A will eventually halt and output 0.

■

R

Remark 11.5 — The Gödel statement (optional). One can extract from the proof of Theorem 11.3 a procedure that for every proof system V, yields a true statement x^* that cannot be proven in V. But Gödel's proof gave a very explicit description of such a statement x^* which is closely related to the "Liar's paradox". That is, Gödel's statement x^* was designed to be true if and only if $\forall_{w\in\{0,1\}^*} V(x,w) = 0$. In other words, it satisfied the following property

$$x^* \text{ is true} \Leftrightarrow x^* \text{ does not have a proof in } V \qquad (11.1)$$

One can see that if x^* is true, then it does not have a proof, but if it is false then (assuming the proof system is sound) then it cannot have a proof, and hence x^* must be both true and unprovable. One might wonder how is it possible to come up with an x^* that satisfies a condition such as (11.1) where the same string x^* appears on both the right-hand side and the left-hand side of the equation. The idea is that the proof of Theorem 11.3 yields a way to transform every statement x into a statement $F(x)$ that is true if and only if x does not have a proof in V. Thus x^* needs to be a *fixed point* of F: a sentence such that $x^* = F(x^*)$. It turns out that we can always find such a fixed point of F. We've already seen this phenomenon in the λ calculus, where the Y combinator maps every F into a fixed point YF of F. This is very related to the idea of programs that can print their own code. Indeed, Scott Aaronson likes to describe Gödel's statement as follows:

> The following sentence repeated twice, the second time in quotes, is not provable in the formal system V. "The following sentence repeated twice, the second time in quotes, is not provable in the formal system V."

In the argument above we actually showed that x^* is *true*, under the assumption that V is sound. Since x^* is true and does not have a proof in V, this means that we cannot carry the above argument in the system V, which means that V cannot prove its own soundness (or even consistency: that there is no proof of both a statement and its negation). Using this idea, it's not hard to get Gödel's second incompleteness theorem, which says that every sufficiently rich V cannot prove

its own consistency. That is, if we formalize the statement c^* that is true if and only if V is consistent (i.e., V cannot prove both a statement and the statement's negation), then c^* cannot be proven in V.

11.3 QUANTIFIED INTEGER STATEMENTS

There is something "unsatisfying" about Theorem 11.3. Sure, it shows there are statements that are unprovable, but they don't feel like "real" statements about math. After all, they talk about *programs* rather than numbers, matrices, or derivatives, or whatever it is they teach in math courses. It turns out that we can get an analogous result for statements such as "there are no positive integers x and y such that $x^2 - 2 = y^7$", or "there are positive integers x, y, z such that $x^2 + y^6 = z^{11}$" that only talk about *natural numbers*. It doesn't get much more "real math" than this. Indeed, the 19th century mathematician Leopold Kronecker famously said that "God made the integers, all else is the work of man." (By the way, the status of the above two statements is unknown.)

To make this more precise, let us define the notion of *quantified integer statements*:

Definition 11.6 — Quantified integer statements. A *quantified integer statement* is a well-formed statement with no unbound variables involving integers, variables, the operators $>, <, \times, +, -, =$, the logical operations $\neg$ (NOT), $\wedge$ (AND), and $\vee$ (OR), as well as quantifiers of the form $\exists_{x \in \mathbb{N}}$ and $\forall_{y \in \mathbb{N}}$ where x, y are variable names.

We often care deeply about determining the truth of quantified integer statements. For example, the statement that Fermat's Last Theorem is true for $n = 3$ can be phrased as the quantified integer statement

$$\neg \exists_{a \in \mathbb{N}} \exists_{b \in \mathbb{N}} \exists_{c \in \mathbb{N}} (a > 0) \wedge (b > 0) \wedge (c > 0) \wedge (a \times a \times a + b \times b \times b = c \times c \times c) \ .$$

The twin prime conjecture, that states that there is an infinite number of numbers p such that both p and $p + 2$ are primes can be phrased as the quantified integer statement

$$\forall_{n \in \mathbb{N}} \exists_{p \in \mathbb{N}} (p > n) \wedge PRIME(p) \wedge PRIME(p + 2)$$

where we replace an instance of $PRIME(q)$ with the statement $(q > 1) \wedge \forall_{a \in \mathbb{N}} \forall_{b \in \mathbb{N}} (a = 1) \vee (a = q) \vee \neg (a \times b = q)$.

The claim (mentioned in Hilbert's quote above) that are infinitely many primes of the form $p = 2^n + 1$ can be phrased as follows:

$$\forall_{n\in\mathbb{N}}\exists_{p\in\mathbb{N}}(p > n) \wedge \mathit{PRIME}(p) \wedge (\forall_{k\in\mathbb{N}}(k \neq 2 \wedge \mathit{PRIME}(k)) \Rightarrow \neg \mathit{DIVIDES}(k, p-1)) \tag{11.2}$$

where $\mathit{DIVIDES}(a, b)$ is the statement $\exists_{c\in\mathbb{N}} b \times c = a$. In English, this corresponds to the claim that for every n there is some $p > n$ such that all of $p - 1$'s prime factors are equal to 2.

R

Remark 11.7 — Syntactic sugar for quantified integer statements. To make our statements more readable, we often use syntactic sugar and so write $x \neq y$ as shorthand for $\neg(x = y)$, and so on. Similarly, the "implication operator" $a \Rightarrow b$ is "syntactic sugar" or shorthand for $\neg a \vee b$, and the "if and only if operator" $a \Leftrightarrow$ is shorthand for $(a \Rightarrow b) \wedge (b \Rightarrow a)$. We will also allow ourselves the use of "macros": plugging in one quantified integer statement in another, as we did with *DIVIDES* and *PRIME* above.

Much of number theory is concerned with determining the truth of quantified integer statements. Since our experience has been that, given enough time (which could sometimes be several centuries) humanity has managed to do so for the statements that it cared enough about, one could (as Hilbert did) hope that eventually we would be able to prove or disprove all such statements. Alas, this turns out to be impossible:

Theorem 11.8 — Gödel's Incompleteness Theorem for quantified integer statements. Let $V : \{0,1\}^* \to \{0,1\}$ a computable purported verification procedure for quantified integer statements. Then either:

- *V is not sound:* There exists a false statement x and a string $w \in \{0,1\}^*$ such that $V(x, w) = 1$.

  or

- *V is not complete:* There exists a true statement x such that for every $w \in \{0,1\}^*$, $V(x, w) = 0$.

Theorem 11.8 is a direct corollary of the following result, just as Theorem 11.3 was a direct corollary of the uncomputability of *HALTONZERO*:

Theorem 11.9 — Uncomputability of quantified integer statements. Let $QIS : \{0,1\}^* \to \{0,1\}$ be the function that given a (string representation of) a quantified integer statement outputs 1 if it is true and 0 if it is false. Then QIS is uncomputable.

Since a quantified integer statement is simply a sequence of symbols, we can easily represent it as a string. For simplicity we will assume that *every* string represents some quantified integer statement, by mapping strings that do not correspond to such a statement to an arbitrary statement such as $\exists_{x \in \mathbb{N}} x = 1$.

Please stop here and make sure you understand why the uncomputability of QIS (i.e., Theorem 11.9) means that there is no sound and complete proof system for proving quantified integer statements (i.e., Theorem 11.8). This follows in the same way that Theorem 11.3 followed from the uncomputability of *HALTONZERO*, but working out the details is a great exercise (see Exercise 11.1)

In the rest of this chapter, we will show the proof of Theorem 11.8, following the outline illustrated in Fig. 11.1.

11.4 DIOPHANTINE EQUATIONS AND THE MRDP THEOREM

Many of the functions people wanted to compute over the years involved solving equations. These have a much longer history than mechanical computers. The Babylonians already knew how to solve some quadratic equations in 2000BC, and the formula for all quadratics appears in the Bakhshali Manuscript that was composed in India around the 3rd century. During the Renaissance, Italian mathematicians discovered generalization of these formulas for cubic and quartic (degrees 3 and 4) equations. Many of the greatest minds of the 17th and 18th century, including Euler, Lagrange, Leibniz and Gauss worked on the problem of finding such a formula for *quintic* equations to no avail, until in the 19th century Ruffini, Abel and Galois showed that no such formula exists, along the way giving birth to *group theory*.

However, the fact that there is no closed-form formula does not mean we can not solve such equations. People have been solving higher degree equations numerically for ages. The Chinese manuscript Jiuzhang Suanshu from the first century mentions such approaches. Solving polynomial equations is by no means restricted only to ancient history or to students' homework. The gradient descent method is the workhorse powering many of the machine

learning tools that have revolutionized Computer Science over the last several years.

But there are some equations that we simply do not know how to solve *by any means*. For example, it took more than 200 years until people succeeded in proving that the equation $a^{11} + b^{11} = c^{11}$ has no solution in integers.[3] The notorious difficulty of so called *Diophantine equations* (i.e., finding *integer* roots of a polynomial) motivated the mathematician David Hilbert in 1900 to include the question of finding a general procedure for solving such equations in his famous list of twenty-three open problems for mathematics of the 20th century. I don't think Hilbert doubted that such a procedure exists. After all, the whole history of mathematics up to this point involved the discovery of ever more powerful methods, and even impossibility results such as the inability to trisect an angle with a straightedge and compass, or the non-existence of an algebraic formula for quintic equations, merely pointed out to the need to use more general methods.

Alas, this turned out not to be the case for Diophantine equations. In 1970, Yuri Matiyasevich, building on a decades long line of work by Martin Davis, Hilary Putnam and Julia Robinson, showed that there is simply *no method* to solve such equations in general:

Theorem 11.10 — MRDP Theorem. Let $DIO : \{0,1\}^* \to \{0,1\}$ be the function that takes as input a string describing a 100-variable polynomial with integer coefficients $P(x_0, \ldots, x_{99})$ and outputs 1 if and only if there exists $z_0, \ldots, z_{99} \in \mathbb{N}$ s.t. $P(z_0, \ldots, z_{99}) = 0$.

Then DIO is uncomputable.

As usual, we assume some standard way to express numbers and text as binary strings. The constant 100 is of course arbitrary; the problem is known to be uncomputable even for polynomials of degree four and at most 58 variables. In fact the number of variables can be reduced to nine, at the expense of the polynomial having a larger (but still constant) degree. See Jones's paper for more about this issue.

R

Remark 11.11 — Active code vs static data. The difficulty in finding a way to distinguish between "code" such as NAND-TM programs, and "static content" such as polynomials is just another manifestation of the phenomenon that *code* is the same as *data*. While a fool-proof solution for distinguishing between the two is inherently impossible, finding heuristics that do a reasonable job keeps many firewall and anti-virus manufacturers very busy (and finding ways to bypass these tools keeps many hackers busy as well).

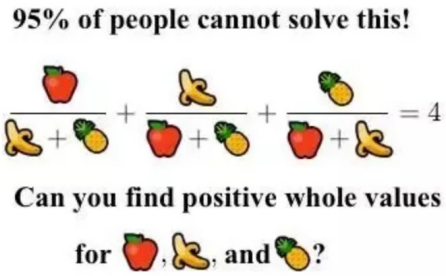


Figure 11.2: Diophantine equations such as finding a positive integer solution to the equation $a(a+b)(a+c)+b(b+a)(b+c)+c(c+a)(c+b) = 4(a+b)(a+c)(b+c)$ (depicted more compactly and whimsically above) can be surprisingly difficult. There are many equations for which we do not know if they have a solution, and there is no algorithm to solve them in general. The smallest solution for this equation has 80 digits! See this Quora post for more information, including the credits for this image.

[3] This is a special case of what's known as "Fermat's Last Theorem" which states that $a^n + b^n = c^n$ has no solution in integers for $n > 2$. This was conjectured in 1637 by Pierre de Fermat but only proven by Andrew Wiles in 1991. The case $n = 11$ (along with all other so called "regular prime exponents") was established by Kummer in 1850.

11.5 HARDNESS OF QUANTIFIED INTEGER STATEMENTS

We will not prove the MRDP Theorem (Theorem 11.10). However, as we mentioned, we will prove the uncomputability of *QIS* (i.e., Theorem 11.9), which is a special case of the MRDP Theorem. The reason is that a Diophantine equation is a special case of a quantified integer statement where the only quantifier is $\exists$. This means that deciding the truth of quantified integer statements is a potentially harder problem than solving Diophantine equations, and so it is potentially *easier* to prove that *QIS* is uncomputable.

If you find the last sentence confusing, it is worthwhile to reread it until you are sure you follow its logic. We are so accustomed to trying to find *solutions* for problems that it can sometimes be hard to follow the arguments for showing that problems are *uncomputable*.

Our proof of the uncomputability of *QIS* (i.e. Theorem 11.9) will, as usual, go by reduction from the Halting problem, but we will do so in two steps:

1. We will first use a reduction from the Halting problem to show that deciding the truth of *quantified mixed statements* is uncomputable. Quantified mixed statements involve both strings and integers. Since quantified mixed statements are a more general concept than quantified integer statements, it is *easier* to prove the uncomputability of deciding their truth.
2. We will then reduce the problem of quantified mixed statements to quantified integer statements.

11.5.1 Step 1: Quantified mixed statements and computation histories

We define *quantified mixed statements* as statements involving not just integers and the usual arithmetic operators, but also *string variables* as well.

Definition 11.12 — Quantified mixed statements. A *quantified mixed statement* is a well-formed statement with no unbound variables involving integers, variables, the operators $>, <, \times, +, -, =$, the logical operations $\neg$ (NOT), $\wedge$ (AND), and $\vee$ (OR), as well as quantifiers of the form $\exists_{x \in \mathbb{N}}$, $\exists_{a \in \{0,1\}^*}$, $\forall_{y \in \mathbb{N}}$, $\forall_{b \in \{0,1\}^*}$ where x, y, a, b are variable names. These also include the operator $|a|$ which returns the length of a string valued variable a, as well as the operator a_i where a is a string-valued variable and i is an integer valued ex-

pression which is true if i is smaller than the length of a and the ith coordinate of a is 1, and is false otherwise.

For example, the true statement that for every string a there is a string b that corresponds to a in reverse order can be phrased as the following quantified mixed statement

$$\forall_{a\in\{0,1\}^*}\exists_{b\in\{0,1\}^*}(|a|=|b|)\wedge(\forall_{i\in\mathbb{N}}i<|a|\Rightarrow(a_i\Leftrightarrow b_{|a|-i}))\ .$$

Quantified mixed statements are more general than quantified integer statements, and so the following theorem is potentially easier to prove than Theorem 11.9:

Theorem 11.13 — Uncomputability of quantified mixed statements. Let $QMS : \{0,1\}^* \to \{0,1\}$ be the function that given a (string representation of) a quantified mixed statement outputs 1 if it is true and 0 if it is false. Then QMS is uncomputable.

Proof Idea:

The idea behind the proof is similar to that used in showing that one-dimensional cellular automata are Turing complete (Theorem 8.7) as well as showing that equivalence (or even "fullness") of context free grammars is uncomputable (Theorem 10.10). We use the notion of a *configuration* of a NAND-TM program as in Definition 8.8. Such a configuration can be thought of as a string α over some large-but-finite alphabet Σ describing its current state, including the values of all arrays, scalars, and the index variable `i`. It can be shown that if α is the configuration at a certain step of the execution and β is the configuration at the next step, then $\beta_j = \alpha_j$ for all j outside of $\{i-1,i,i+1\}$ where i is the value of `i`. In particular, every value β_j is simply a function of $\alpha_{j-1,j,j+1}$. Using these observations we can write a *quantified mixed statement* $NEXT(\alpha,\beta)$ that will be true if and only if β is the configuration encoding the next step after α. Since a program P halts on input x if and only if there is a sequence of configurations $\alpha^0,\ldots,\alpha^{t-1}$ (known as a *computation history*) starting with the initial configuration with input x and ending in a halting configuration, we can define a quantified mixed statement to determine if there is such a statement by taking a universal quantifier over all strings H (for *history*) that encode a tuple $(\alpha^0,\alpha^1,\ldots,\alpha^{t-1})$ and then checking that α^0 and α^{t-1} are valid starting and halting configurations, and that $NEXT(\alpha^j,\alpha^{j+1})$ is true for every $j\in\{0,\ldots,t-2\}$.

⋆

Proof of Theorem 11.13. The proof is obtained by a reduction from the Halting problem. Specifically, we will use the notion of a *configuration* of a Turing machines (Definition 8.8) that we have seen in the

context of proving that one dimensional cellular automata are Turing complete. We need the following facts about configurations:

- For every Turing machine M, there is a finite alphabet Σ, and a *configuration* of M is a string $\alpha \in \Sigma^*$.
- A configuration α encodes all the state of the program at a particular iteration, including the array, scalar, and index variables.
- If α is a configuration, then $\beta = NEXT_P(\alpha)$ denotes the configuration of the computation after one more iteration. β is a string over Σ of length either $|\alpha|$ or $|\alpha|+1$, and every coordinate of β is a function of just three coordinates in α. That is, for every $j \in \{0,\ldots,|\beta|-1\}$, $\beta_j = MAP_P(\alpha_{j-1},\alpha_j,\alpha_{j+1})$ where $MAP_P : \Sigma^3 \to \Sigma$ is some function depending on P.
- There are simple conditions to check whether a string α is a valid starting configuration corresponding to an input x, as well as to check whether a string α is a halting configuration. In particular these conditions can be phrased as quantified mixed statements.
- A program M halts on input x if and only if there exists a sequence of configurations $H = (\alpha^0,\alpha^1,\ldots,\alpha^{T-1})$ such that **(i)** α^0 is a valid starting configuration of M with input x, **(ii)** α^{T-1} is a valid halting configuration of P, and **(iii)** $\alpha^{i+1} = NEXT_P(\alpha^i)$ for every $i \in \{0,\ldots,T-2\}$.

We can encode such a sequence H of configuration as a binary string. For concreteness, we let $\ell = \lceil \log(|\Sigma|+1) \rceil$ and encode each symbol σ in $\Sigma \cup \{";"\}$ by a string in $\{0,1\}^\ell$. We use ";" as a "separator" symbol, and so encode $H = (\alpha^0,\alpha^1,\ldots,\alpha^{T-1})$ as the concatenation of the encodings of each configuration, using ";" to separate the encoding of α^i and α^{i+1} for every $i \in [T]$. In particular for every Turing machine M, M halts on the input 0 if and only if the following statement φ_M is true

$\exists_{H \in \{0,1\}^*} H$ encodes halting configuration sequence starting with input 0 .

If we can encode the statement φ_M as a quantified mixed statement then, since φ_M is true if and only if $HALTONZERO(M) = 1$, this would reduce the task of computing *HALTONZERO* to computing *QMS*, and hence imply (using Theorem 9.9) that *QMS* is uncomputable, completing the proof. Indeed, φ_M can be encoded as a quantified mixed statement for the following reasons:

1. Let $\alpha,\beta \in \{0,1\}^*$ be two strings that encode configurations of M. We can define a quantified mixed predicate $NEXT(\alpha,\beta)$ that is true

if and only if $\beta = NEXT_M(\alpha)$ (i.e., β encodes the configuration obtained by proceeding from α in one computational step). Indeed $NEXT(\alpha,\beta)$ is true if **for every** $i \in \{0,\ldots,|\beta|\}$ which is a multiple of ℓ, $\beta_{i,\ldots,i+\ell-1} = MAP_M(\alpha_{i-\ell,\cdots,i+2\ell-1})$ where $MAP_M : \{0,1\}^{3\ell} \to \{0,1\}^{\ell}$ is the finite function above (identifying elements of Σ with their encoding in $\{0,1\}^{\ell}$). Since MAP_M is a finite function, we can express it using the logical operations *AND*,*OR*, *NOT* (for example by computing MAP_M with *NAND*'s).

2. Using the above we can now write the condition that **for every** substring of H that has the form $\alpha ENC(;)\beta$ with $\alpha,\beta \in \{0,1\}^{\ell}$ and $ENC(;)$ being the encoding of the separator ";", it holds that $NEXT(\alpha,\beta)$ is true.

3. Finally, if α^0 is a binary string encoding the initial configuration of M on input 0, checking that the first $|\alpha^0|$ bits of H equal α_0 can be expressed using *AND*,*OR*, and *NOT*'s. Similarly checking that the last configuration encoded by H corresponds to a state in which M will halt can also be expressed as a quantified statement.

Together the above yields a computable procedure that maps every Turing machine M into a quantified mixed statement φ_M such that $HALTONZERO(M) = 1$ if and only if $QMS(\varphi_M) = 1$. This reduces computing *HALTONZERO* to computing *QMS*, and hence the uncomputability of *HALTONZERO* implies the uncomputability of *QMS*. ■

R

Remark 11.14 — Alternative proofs. There are several other ways to show that *QMS* is uncomputable. For example, we can express the condition that a 1-dimensional cellular automaton eventually writes a "1" to a given cell from a given initial configuration as a quantified mixed statement over a string encoding the history of all configurations. We can then use the fact that cellular automatons can simulate Turing machines (Theorem 8.7) to reduce the halting problem to *QMS*. We can also use other well known uncomputable problems such as tiling or the post correspondence problem. Exercise 11.5 and Exercise 11.6 explore two alternative proofs of Theorem 11.13.

11.5.2 Step 2: Reducing mixed statements to integer statements

We now show how to prove Theorem 11.9 using Theorem 11.13. The idea is again a proof by reduction. We will show a transformation of every quantified mixed statement φ into a quantified *integer* statement

ξ that does not use string-valued variables such that φ is true if and only if ξ is true.

To remove string-valued variables from a statement, we encode every string by a pair integer. We will show that we can encode a string $x \in \{0,1\}^*$ by a pair of numbers $(X, n) \in \mathbb{N}$ s.t.

- $n = |x|$
- There is a quantified integer statement $COORD(X, i)$ that for every $i < n$, will be true if $x_i = 1$ and will be false otherwise.

This will mean that we can replace a "for all" quantifier over strings such as $\forall_{x\in\{0,1\}^*}$ with a pair of quantifiers over *integers* of the form $\forall_{X\in\mathbb{N}}\forall_{n\in\mathbb{N}}$ (and similarly replace an existential quantifier of the form $\exists_{x\in\{0,1\}^*}$ with a pair of quantifiers $\exists_{X\in\mathbb{N}}\exists_{n\in\mathbb{N}}$) . We can then replace all calls to $|x|$ by n and all calls to x_i by $COORD(X, i)$. This means that if we are able to define *COORD* via a quantified integer statement, then we obtain a proof of Theorem 11.9, since we can use it to map every mixed quantified statement φ to an equivalent quantified integer statement ξ such that ξ is true if and only if φ is true, and hence $QMS(\varphi) = QIS(\xi)$. Such a procedure implies that the task of computing *QMS* reduces to the task of computing *QIS*, which means that the uncomputability of *QMS* implies the uncomputability of *QIS*.

The above shows that proof of Theorem 11.9 all boils down to finding the right encoding of strings as integers, and the right way to implement *COORD* as a quantified integer statement. To achieve this we use the following technical result :

Lemma 11.15 — Constructible prime sequence. There is a sequence of prime numbers $p_0 < p_1 < p_2 < \cdots$ such that there is a quantified integer statement $PSEQ(p, i)$ that is true if and only if $p = p_i$.

Using Lemma 11.15 we can encode a $x \in \{0,1\}^*$ by the numbers (X, n) where $X = \prod_{x_i=1} p_i$ and $n = |x|$. We can then define the statement $COORD(X, i)$ as

$$COORD(X, i) = \exists_{p\in\mathbb{N}} PSEQ(p, i) \wedge DIVIDES(p, X)$$

where $DIVIDES(a, b)$, as before, is defined as $\exists_{c\in\mathbb{N}} a \times c = b$. Note that indeed if X, n encodes the string $x \in \{0,1\}^*$, then for every $i < n$, $COORD(X, i) = x_i$, since p_i divides X if and only if $x_i = 1$.

Thus all that is left to conclude the proof of Theorem 11.9 is to prove Lemma 11.15, which we now proceed to do.

Proof. The sequence of prime numbers we consider is the following: We fix C to be a sufficiently large constant ($C = 2^{2^{34}}$ will do) and define p_i to be the smallest prime number that is in the interval $[(i + C)^3 + 1, (i + C + 1)^3 - 1]$. It is known that there exists such a prime

number for every $i \in \mathbb{N}$. Given this, the definition of $PSEQ(p,i)$ is simple:

$$(p > (i+C)\times(i+C)\times(i+C)) \wedge (p < (i+C+1)\times(i+C+1)\times(i+C+1) \wedge PRIME(p) \wedge (\forall_{p'} \neg PRIME(p') \vee (p' \leq (i+C)\times(i+C)\times(i+C$$

We leave it to the reader to verify that $PSEQ(p,i)$ is true iff $p = p_i$. ■

To sum up we have shown that for every quantified mixed statement φ, we can compute a quantified integer statement ξ such that $QMS(\varphi) = 1$ if and only if $QIS(\xi) = 1$. Hence the uncomputability of QMS (Theorem 11.13) implies the uncomputability of QIS, completing the proof of Theorem 11.9, and so also the proof of Gödel's Incompleteness Theorem for quantified integer statements (Theorem 11.8).

Chapter Recap

- Uncomputable functions include also functions that seem to have nothing to do with NAND-TM programs or other computational models such as determining the satisfiability of Diophantine equations.
- This also implies that for any sound proof system (and in particular every finite axiomatic system) S, there are interesting statements X (namely of the form "$F(x) = 0$" for an uncomputable function F) such that S is not able to prove either X or its negation.

11.6 EXERCISES

Exercise 11.1 — Gödel's Theorem from uncomputability of QIS. Prove Theorem 11.8 using Theorem 11.9. ■

Exercise 11.2 — Proof systems and uncomputability. Let $FINDPROOF : \{0,1\}^* \to \{0,1\}$ be the following function. On input a Turing machine V (which we think of as the verifying algorithm for a proof system) and a string $x \in \{0,1\}^*$, $FINDPROOF(V,x) = 1$ if and only if there exists $w \in \{0,1\}^*$ such that $V(x,w) = 1$.

1. Prove that $FINDPROOF$ is uncomputable.
2. Prove that there exists a Turing machine V such that V *halts on every input* x, v but the function $FINDPROOF_V$ defined as $FINDPROOF_V(x) = FINDPROOF(V,x)$ is uncomputable. See footnote for hint.[4]

[4] *Hint:* think of x as saying "Turing machine M halts on input u" and w being a proof that is the number of steps that it will take for this to happen. Can you find an always-halting V that will verify such statements?

Exercise 11.3 — Expression for floor. Let $FSQRT(n,m) = \forall_{j\in\mathbb{N}}((j \times j) > m) \vee (j \leq n)$. Prove that $FSQRT(n,m)$ is true if and only if $n = \lfloor\sqrt{m}\rfloor$.

Exercise 11.4 — axiomatic proof systems. For every representation of logical statements as strings, we can define an axiomatic proof system to consist of a finite set of strings A and a finite set of rules $I_0, \ldots, I_{m-1}$ with $I_j : (\{0,1\}^*)^{k_j} \to \{0,1\}^*$ such that a proof $(s_1, \ldots, s_n)$ that s_n is true is valid if for every i, either $s_i \in A$ or is some $j \in [m]$ and are $i_1, \ldots, i_{k_j} < i$ such that $s_i = I_j(s_{i_1}, \ldots, i_{k_j})$. A system is *sound* if whenever there is no false s such that there is a proof that s is true. Prove that for every uncomputable function $F : \{0,1\}^* \to \{0,1\}$ and every sound axiomatic proof system S (that is characterized by a finite number of axioms and inference rules), there is some input x for which the proof system S is not able to prove neither that $F(x) = 0$ nor that $F(x) \neq 0$.

Exercise 11.5 — Post Corrrespondence Problem. In the Post Correspondence Problem the input is a set $S = \{(\alpha^0, \beta^0), \ldots, (\beta^{c-1}, \beta^{c-1})\}$ where each α^i and β^j is a string in $\{0,1\}^*$. We say that $PCP(S) = 1$ if and only if there exists a list $(\alpha_0, \beta_0), \ldots, (\alpha_{m-1}, \beta_{m-1})$ of pairs in S such that

$$\alpha_0\alpha_1 \cdots \alpha_{m-1} = \beta_0\beta_1 \cdots \beta_{m-1} .$$

(We can think of each pair $(\alpha, \beta) \in S$ as a "domino tile" and the question is whether we can stack a list of such tiles so that the top and the bottom yield the same string.) It can be shown that the *PCP* is uncomputable by a fairly straightforward though somewhat tedious proof (see for example the Wikipedia page for the Post Correspondence Problem or Section 5.2 in [Sip97]).

Use this fact to provide a direct proof that *QMS* is uncomputable by showing that there exists a computable map $R : \{0,1\}^* \to \{0,1\}^*$ such that $PCP(S) = QMS(R(S))$ for every string S encoding an instance of the post correspondence problem.

Exercise 11.6 — Uncomputability of puzzle. Let $PUZZLE : \{0,1\}^* \to \{0,1\}$ be the problem of determining, given a finite collection of types of "puzzle pieces", whether it is possible to put them together in a rectangle, see Fig. 11.3. Formally, we think of such a collection as a finite set Σ (see Fig. 11.3). We model the criteria as to which pieces "fit together" by a pair of finite function $match_{\updownarrow}, match_{\leftrightarrow} : \Sigma^2 \to \{0,1\}$ such that a piece a fits above a piece b if and only if $match_{\updownarrow}(a,b) = 1$ and a piece c fits to the left of a piece d if and only if $match_{\leftrightarrow}(c,d) = 1$. To model

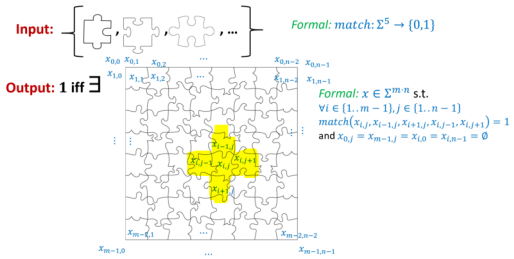


Figure 11.3: In the *puzzle problem*, the input can be thought of as a finite collection Σ of *types of puzzle pieces* and the goal is to find out whether or not find a way to arrange pieces from these types in a rectangle. Formally, we model the input as a pair of functions $match_{\leftrightarrow}, match_{\updownarrow} : \Sigma^2 \to \{0,1\}$ that such that $match_{\leftrightarrow}(left, right) = 1$ (respectively $match_{\updownarrow}(up, down) = 1$) if the pair of pieces are compatible when placed in their respective positions. We assume Σ contains a special symbol $\varnothing$ corresponding to having no piece, and an arrangement of puzzle pieces by an $(m-2) \times (n-2)$ rectangle is modeled by a string $x \in \Sigma^{m \cdot n}$ whose "outer coordinates'' are $\varnothing$ and such that for every $i \in [n-1], j \in [m-1]$, $match_{\updownarrow}(x_{i,j}, x_{i+1,j}) = 1$ and $match_{\leftrightarrow}(x_{i,j}, x_{i,j+1}) = 1$.

the "straight edge" pieces that can be placed next to a "blank spot" we assume that Σ contains the symbol $\varnothing$ and the matching functions are defined accordingly. A *square tiling* of Σ is an $m \times n$ long string $x \in \Sigma^{mn}$, such that for every $i \in \{1, \ldots, m-2\}$ and $j \in \{1, \ldots, n-2\}$, $match(x_{i,j}, x_{i-1,j}, x_{i+1,j}, x_{i,j-1}, x_{i,j+1}) = 1$ (i.e., every "internal pieve" fits in with the pieces adjacent to it). We also require all of the "outer pieces" (i.e., $x_{i,j}$ where $i \in \{0, m-1\}$ of $j \in \{0, n-1\}$) are "blank" or equal to $\varnothing$. The function *PUZZLE* takes as input a string describing the set Σ and the function $match$ and outputs 1 if and only if there is some square tiling of Σ: some not all blank string $x \in \Sigma^{mn}$ satisfying the above condition.

1. Prove that *PUZZLE* is uncomputable.

2. Give a reduction from *PUZZLE* to *QMS*.

■

Exercise 11.7 — MRDP exercise. The MRDP theorem states that the problem of determining, given a k-variable polynomial p with integer coefficients, whether there exists integers $x_0, \ldots, x_{k-1}$ such that $p(x_0, \ldots, x_{k-1}) = 0$ is uncomputable. Consider the following *quadratic integer equation problem*: the input is a list of polynomials $p_0, \ldots, p_{m-1}$ over k variables with integer coefficients, where each of the polynomials is of degree at most two (i.e., it is a *quadratic* function). The goal is to determine whether there exist integers $x_0, \ldots, x_{k-1}$ that solve the equations $p_0(x) = \cdots = p_{m-1}(x) = 0$.

Use the MRDP Theorem to prove that this problem is uncomputable. That is, show that the function $QUADINTEQ : \{0,1\}^* \to \{0,1\}$ is uncomputable, where this function gets as input a string describing the polynomials $p_0, \ldots, p_{m-1}$ (each with integer coefficients and degree at most two), and outputs 1 if and only if there exists $x_0, \ldots, x_{k-1} \in \mathbb{Z}$ such that for every $i \in [m]$, $p_i(x_0, \ldots, x_{k-1}) = 0$. See footnote for hint[5]

■

Exercise 11.8 — The Busy Beaver problem. In this question we define the NAND-TM variant of the busy beaver function.

1. We define the function $T : \{0,1\}^* \to \mathbb{N}$ as follows: for every string $P \in \{0,1\}^*$, if P represents a NAND-TM program such that when P is executed on the input 0 (i.e., the string of length 1 that is simply 0), a total of M lines are executed before the program halts, then $T(P) = M$. Otherwise (if P does not represent a NAND-TM program, or it is a program that does not halt on 0), $T(P) = 0$. Prove that T is uncomputable.

[5] You can replace the equation $y = x^4$ with the pair of equations $y = z^2$ and $z = x^2$. Also, you can replace the equation $w = x^6$ with the three equations $w = yu$, $y = x^4$ and $u = x^2$.

2. Let $TOWER(n)$ denote the number $\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n \text{ times}}$ (that is, a "tower of powers of two" of height n). To get a sense of how fast this function grows, $TOWER(1) = 2$, $TOWER(2) = 2^2 = 4$, $TOWER(3) = 2^{2^2} = 16$, $TOWER(4) = 2^{16} = 65536$ and $TOWER(5) = 2^{65536}$ which is about 10^{20000}. $TOWER(6)$ is already a number that is too big to write even in scientific notation. Define $NBB : \mathbb{N} \to \mathbb{N}$ (for "NAND-TM Busy Beaver") to be the function $NBB(n) = \max_{P \in \{0,1\}^n} T(P)$ where $T : \mathbb{N} \to \mathbb{N}$ is the function defined in Item 1. Prove that NBB grows *faster* than $TOWER$, in the sense that $TOWER(n) = o(NBB(n))$ (i.e., for every $\epsilon > 0$, there exists n_0 such that for every $n > n_0$, $TOWER(n) < \epsilon \cdot NBB(n)$.).[6]

[6] You will not need to use very specific properties of the $TOWER$ function in this exercise. For example, $NBB(n)$ also grows faster than the Ackerman function. You might find Aaronson's blog post on the same topic to be quite interesting, and relevant to this book at large. If you like it then you might also enjoy this piece by Terence Tao.

11.7 BIBLIOGRAPHICAL NOTES

As mentioned before, Gödel, Escher, Bach [Hof99] is a highly recommended book covering Gödel's Theorem. A classic popular science book about Fermat's Last Theorem is [Sin97].

Cantor's are used for both Turing and Gödel's theorems. In a twist of fate, using techniques originating from the works of Gödel and Turing, Paul Cohen showed in 1963 that Cantor's *Continuum Hypothesis* is independent of the axioms of set theory, which means that neither it nor its negation is provable from these axioms and hence in some sense can be considered as "neither true nor false" (see [Coh08]). The Continuum Hypothesis is the conjecture that for every subset S of $\mathbb{R}$, either there is a one-to-one and onto map between S and $\mathbb{N}$ or there is a one-to-one and onto map between S and $\mathbb{R}$. It was conjectured by Cantor and listed by Hilbert in 1900 as one of the most important problems in mathematics. See also the non-conventional survey of Shelah [She03]. See here for recent progress on a related question.

Thanks to Alex Lombardi for pointing out an embarrassing mistake in the description of Fermat's Last Theorem. (I said that it was open for exponent 11 before Wiles' work.)

III

EFFICIENT ALGORITHMS

Learning Objectives:

- **Describe at a high level some interesting computational problems.**
- **The difference between polynomial and exponential time.**
- **Examples of techniques for obtaining efficient algorithms**
- **Examples of how seemingly small differences in problems can potentially make huge differences in their computational complexity.**

12
Efficient computation: An informal introduction

> *"The problem of distinguishing prime numbers from composite and of resolving the latter into their prime factors is ... one of the most important and useful in arithmetic ... Nevertheless we must confess that all methods ... are either restricted to very special cases or are so laborious ... they try the patience of even the practiced calculator ... and do not apply at all to larger numbers."*, Carl Friedrich Gauss, 1798

> *"For practical purposes, the difference between algebraic and exponential order is often more crucial than the difference between finite and non-finite."*, Jack Edmunds, "Paths, Trees, and Flowers", 1963

> *"What is the most efficient way to sort a million 32-bit integers?"*, Eric Schmidt to Barack Obama, 2008

> *"I think the bubble sort would be the wrong way to go."*, Barack Obama.

So far we have been concerned with which functions are *computable* and which ones are not. In this chapter we look at the finer question of the *time* that it takes to compute functions, as a *function of their input length*. Time complexity is extremely important to both the theory and practice of computing, but in introductory courses, coding interviews, and software development, terms such as "$O(n)$ running time" are often used in an informal way. People don't have a precise definition of what a linear-time algorithm is, but rather assume that "they'll know it when they see it". In this book we will define running time precisely, using the mathematical models of computation we developed in the previous chapters. This will allow us to ask (and sometimes answer) questions such as:

- "Is there a function that can be computed in $O(n^2)$ time but not in $O(n)$ time?"
- "Are there natural problems for which the *best* algorithm (and not just the *best known*) requires $2^{\Omega(n)}$ time?"

Big Idea 16 The running time of an algorithm is not a *number*, it is a *function* of the length of the input.

This chapter: A non-mathy overview

In this chapter, we informally survey examples of computational problems. For some of these problems we know efficient (i.e., $O(n^c)$-time for a small constant c) algorithms, and for others the best known algorithms are *exponential*. We present these examples to get a feel as to the kinds of problems that lie on each side of this divide and also see how sometimes seemingly minor changes in problem formulation can make the (known) complexity of a problem "jump" from polynomial to exponential. We do not formally define the notion of running time in this chapter, but use the same "I know it when I see it" notion of an $O(n)$ or $O(n^2)$ time algorithms as the one you've seen in introduction to computer science courses. We will see the precise definition of running time (using Turing machines and RAM machines / NAND-RAM) in Chapter 13.

While the difference between $O(n)$ and $O(n^2)$ time can be crucial in practice, in this book we focus on the even bigger difference between *polynomial* and *exponential* running time. As we will see, the difference between polynomial versus exponential time is typically *insensitive* to the choice of the particular computational model, a polynomial-time algorithm is still polynomial whether you use Turing machines, RAM machines, or parallel cluster as your model of computation, and similarly an exponential-time algorithm will remain exponential in all of these platforms. One of the interesting phenomena of computing is that there is often a kind of a "threshold phenomenon" or "zero-one law" for running time. Many natural problems can either be solved in *polynomial* running time with a *not-too-large exponent* (e.g., something like $O(n^2)$ or $O(n^3)$), or require *exponential* (e.g., at least $2^{\Omega(n)}$ or $2^{\Omega(\sqrt{n})}$) time to solve. The reasons for this phenomenon are still not fully understood, but some light on it is shed by the concept of *NP completeness*, which we will see in Chapter 15.

This chapter is merely a tiny sample of the landscape of computational problems and efficient algorithms. If you want to explore the field of algorithms and data structures more deeply (which I very much hope you do!), the bibliographical notes contain references to some excellent texts, some of which are available freely on the web.

R

Remark 12.1 — Relations between parts of this book. **Part I** of this book contained a *quantitative study* of computation of *finite functions*. We asked what are the resources (in terms of gates of Boolean circuits or lines in straight-line programs) required to compute various finite functions.

Part II of the book contained a *qualitative study* of computation of *infinite functions* (i.e., functions of *unbounded input length*). In that part we asked the *qualitative question* of whether or not a function is computable at all, regardless of the number of operations.

Part III of the book, beginning with this chapter, merges the two approaches and contains a *quantitative study* of computation of *infinite functions*. In this part we ask how do resources for computing a function *scale* with the length of the input. In Chapter 13 we define the notion of running time, and the class **P** of functions that can be computed using a number of steps that scales *polynomially* with the input length. In Section 13.6 we will relate this class to the models of Boolean circuits and straightline programs that we studied in Part I.

12.1 PROBLEMS ON GRAPHS

In this chapter we discuss several examples of important computational problems. Many of the problems will involve *graphs*. We have already encountered graphs before (see Section 1.4.4) but now quickly recall the basic notation. A graph G consists of a set of *vertices* V and *edges* E where each edge is a pair of vertices. We typically denote by n the number of vertices (and in fact often consider graphs where the set of vertices V equals the set $[n]$ of the integers between 0 and $n-1$). In a *directed* graph, an edge is an ordered pair (u,v), which we sometimes denote as $\overrightarrow{u\,v}$. In an *undirected* graph, an edge is an unordered pair (or simply a set) $\{u,v\}$ which we sometimes denote as $\overline{u\,v}$ or $u \sim v$. An equivalent viewpoint is that an undirected graph corresponds to a directed graph satisfying the property that whenever the edge $\overrightarrow{u\,v}$ is present then so is the edge $\overrightarrow{v\,u}$. In this chapter we restrict our attention to graphs that are undirected and simple (i.e., containing no parallel edges or self-loops). Graphs can be represented either in the *adjacency list* or *adjacency matrix* representation. We can transform between these two representations using $O(n^2)$ operations, and hence for our purposes we will mostly consider them as equivalent.

Graphs are so ubiquitous in computer science and other sciences because they can be used to model a great many of the data that we encounter. These are not just the "obvious" data such as the road

network (which can be thought of as a graph of whose vertices are locations with edges corresponding to road segments), or the web (which can be thought of as a graph whose vertices are web pages with edges corresponding to links), or social networks (which can be thought of as a graph whose vertices are people and the edges correspond to friend relation). Graphs can also denote correlations in data (e.g., graph of observations of features with edges corresponding to features that tend to appear together), causal relations (e.g., gene regulatory networks, where a gene is connected to gene products it derives), or the state space of a system (e.g., graph of configurations of a physical system, with edges corresponding to states that can be reached from one another in one step).

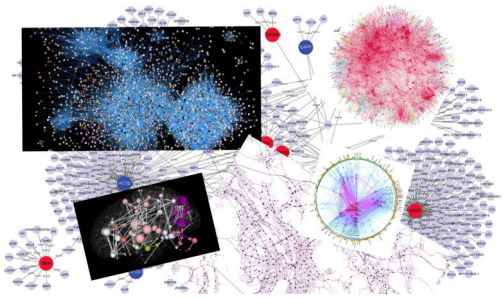


Figure 12.1: Some examples of graphs found on the Internet.

12.1.1 Finding the shortest path in a graph

The *shortest path problem* is the task of finding, given a graph $G = (V, E)$ and two vertices $s, t \in V$, the length of the shortest path between s and t (if such a path exists). That is, we want to find the smallest number k such that there are vertices $v_0, v_1, \ldots, v_k$ with $v_0 = s$, $v_k = t$ and for every $i \in \{0, \ldots, k-1\}$ an edge between v_i and v_{i+1}. Formally, we define $MINPATH : \{0,1\}^* \to \{0,1\}^*$ to be the function that on input a triple (G, s, t) (represented as a string) outputs the number k which is the length of the shortest path in G between s and t or a string representing `no path` if no such path exists. (In practice people often want to also find the actual path and not just its length; it turns out that the algorithms to compute the length of the path often yield the actual path itself as a byproduct, and so everything we say about the task of computing the length also applies to the task of finding the path.)

If each vertex has at least two neighbors then there can be an *exponential* number of paths from s to t, but fortunately we do not have to enumerate them all to find the shortest path. We can find the shortest path using a breadth first search (BFS), enumerating s's neighbors, and then neighbors' neighbors, etc.. in order. If we maintain the neighbors in a list we can perform a BFS in $O(n^2)$ time, while using a *queue* we can do this in $O(m)$ time.[1] Dijkstra's algorithm is a well-known generalization of BFS to *weighted* graphs. More formally, the algorithm for computing the function *MINPATH* is described in Algorithm 12.2.

[1] A *queue* is a data structure for storing a list of elements in "First In First Out (FIFO)" order. Each "pop" operation removes an element from the queue in the order that they were "pushed" into it; see the Wikipedia page.

Algorithm 12.2 — Shortest path via BFS.

Input: Graph $G = (V, E)$ and vertices $s, t \in V$. Assume $V = [n]$.

Output: Length k of shortest path from s to t or ∞ if no such path exists.

```
Let D be length-n array.
Set D[s] = 0 and D[i] = ∞ for all i ∈ [n] \ {s}.
Initialize queue Q to contain s.
while Q non empty do
    Pop v from Q
    if v = t then
        return D[v]
    end if
    for u neighbor of v with D[u] = ∞ do
        Set D[u] ← D[v] + 1
        Add u to Q.
    end for
end while
return ∞
```

Since we only add to the queue vertices w with $D[w] = \infty$ (and then immediately set $D[w]$ to an actual number), we never push to the queue a vertex more than once, and hence the algorithm makes at most n "push" and "pop" operations. For each vertex v, the number of times we run the inner loop is equal to the *degree* of v and hence the total running time is proportional to the sum of all degrees which equals twice the number m of edges. Algorithm 12.2 returns the correct answer since the vertices are added to the queue in the order of their distance from s, and hence we will reach t after we have explored all the vertices that are closer to s than t.

R

Remark 12.3 — On data structures. If you've ever taken an algorithms course, you have probably encountered many *data structures* such as **lists**, **arrays**, **queues**, **stacks**, **heaps**, **search trees**, **hash tables** and many more. Data structures are extremely important in computer science, and each one of those offers different tradeoffs between overhead in storage, operations supported, cost in time for each operation, and more. For example, if we store n items in a list, we will need a linear (i.e., $O(n)$ time) scan to retrieve an element, while we achieve the same operation in $O(1)$ time if we used a hash table. However, when we only care about polynomial-time algorithms, such factors of $O(n)$ in the running time will not make much differ-

ence. Similarly, if we don't care about the difference between $O(n)$ and $O(n^2)$, then it doesn't matter if we represent graphs as adjacency lists or adjacency matrices. Hence we will often describe our algorithms at a very high level, without specifying the particular data structures that are used to implement them. However, it will always be clear that there exists *some* data structure that is sufficient for our purposes.

12.1.2 Finding the longest path in a graph

The *longest path problem* is the task of finding the length of the *longest* simple (i.e., non-intersecting) path between a given pair of vertices s and t in a given graph G. If the graph is a road network, then the longest path might seem less motivated than the shortest path (unless you are the kind of person that always prefers the "scenic route"). But graphs can and are used to model a variety of phenomena, and in many such cases finding the longest path (and some of its variants) can be very useful. In particular, finding the longest path is a generalization of the famous Hamiltonian path problem which asks for a *maximally long* simple path (i.e., path that visits all n vertices once) between s and t, as well as the notorious traveling salesman problem (TSP) of finding (in a weighted graph) a path visiting all vertices of cost at most w. TSP is a classical optimization problem, with applications ranging from planning and logistics to DNA sequencing and astronomy.

Surprisingly, while we can find the shortest path in $O(m)$ time, there is no known algorithm for the *longest path problem* that significantly improves on the trivial "exhaustive search" or "brute force" algorithm that enumerates all the exponentially many possibilities for such paths. Specifically, the best known algorithms for the longest path problem take $O(c^n)$ time for some constant $c > 1$. (At the moment the best record is $c \sim 1.65$ or so; even obtaining an $O(2^n)$ time bound is not that simple, see Exercise 12.1.)

12.1.3 Finding the minimum cut in a graph

Given a graph $G = (V, E)$, a *cut* of G is a subset $S \subseteq V$ such that S is neither empty nor is it all of V. The edges cut by S are those edges where one of their endpoints is in S and the other is in $\overline{S} = V \setminus S$. We denote this set of edges by $E(S, \overline{S})$. If $s, t \in V$ are a pair of vertices then an s, t *cut* is a cut such that $s \in S$ and $t \in \overline{S}$ (see Fig. 12.3). The *minimum* s, t *cut problem* is the task of finding, given s and t, the minimum number k such that there is an s, t cut cutting k edges (the problem is also sometimes phrased as finding the set that achieves this minimum; it turns out that algorithms to compute the number often yield the set as well). Formally, we define $MINCUT : \{0,1\}^* \rightarrow$

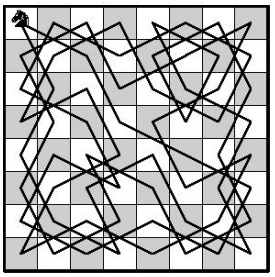


Figure 12.2: A *knight's tour* can be thought of as a maximally long path on the graph corresponding to a chessboard where we put an edge between any two squares that can be reached by one step via a legal knight move.

$\{0,1\}^*$ to be the function that on input a string representing a triple $(G = (V, E), s, t)$ of a graph and two vertices, outputs the minimum number k such that there exists a set $S \subseteq V$ with $s \in S, t \notin S$ and $|E(S, \overline{S})| = k$.

Computing minimum s, t cuts is useful in many applications since minimum cuts often correspond to *bottlenecks*. For example, in a communication or railroad network the minimum cut between s and t corresponds to the smallest number of edges that, if dropped, will disconnect s from t. (This was actually the original motivation for this problem; see Section 12.6.) Similar applications arise in scheduling and planning. In the setting of image segmentation, one can define a graph whose vertices are pixels and whose edges correspond to neighboring pixels of distinct colors. If we want to separate the foreground from the background then we can pick (or guess) a foreground pixel s and background pixel t and ask for a minimum cut between them.

The naive algorithm for computing *MINCUT* will check all 2^n possible subsets of an n-vertex graph, but it turns out we can do much better than that. As we've seen in this book time and again, there is more than one algorithm to compute the same function, and some of those algorithms might be more efficient than others. Luckily the minimum cut problem is one of those cases. In particular, as we will see in the next section, there are algorithms that compute *MINCUT* in time which is *polynomial* in the number of vertices.

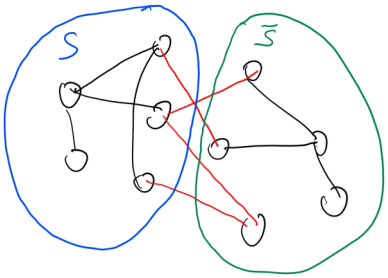


Figure 12.3: A *cut* in a graph $G = (V, E)$ is simply a subset S of its vertices. The edges that are *cut* by S are all those whose one endpoint is in S and the other one is in $\overline{S} = V \setminus S$. The cut edges are colored red in this figure.

12.1.4 Min-Cut Max-Flow and Linear programming

We can obtain a polynomial-time algorithm for computing *MINCUT* using the Max-Flow Min-Cut Theorem. This theorem says that the minimum cut between s and t equals the maximum amount of *flow* we can send from s to t, if every edge has unit capacity. Specifically, imagine that every edge of the graph corresponded to a pipe that could carry one unit of fluid per one unit of time (say 1 liter of water per second). The *maximum* s, t *flow* is the maximum units of water that we could transfer from s to t over these pipes. If there is an s, t cut of k edges, then the maximum flow is at most k. The reason is that such a cut S acts as a "bottleneck" since at most k units can flow from S to its complement at any given unit of time. This means that the maximum s, t flow is always *at most* the value of the minimum s, t cut. The surprising and non-trivial content of the Max-Flow Min-Cut Theorem is that the maximum flow is also *at least* the value of the minimum cut, and hence computing the cut is the same as computing the flow.

The Max-Flow Min-Cut Theorem reduces the task of computing a minimum cut to the task of computing a *maximum flow*. However, this still does not show how to compute such a flow. The Ford-Fulkerson

Algorithm is a direct way to compute a flow using incremental improvements. But computing flows in polynomial time is also a special case of a much more general tool known as linear programming.

A *flow* on a graph G of m edges can be modeled as a vector $x \in \mathbb{R}^m$ where for every edge e, x_e corresponds to the amount of water per time-unit that flows on e. We think of an edge e as an ordered pair (u, v) (we can choose the order arbitrarily) and let x_e be the amount of flow that goes from u to v. (If the flow is in the other direction then we make x_e negative.) Since every edge has capacity one, we know that $-1 \leq x_e \leq 1$ for every edge e. A valid flow has the property that the amount of water leaving the source s is the same as the amount entering the sink t, and that for every other vertex v, the amount of water entering and leaving v is the same.

Mathematically, we can write these conditions as follows:

$$\begin{aligned} \sum_{e \ni s} x_e + \sum_{e \ni t} x_e &= 0 \\ \sum_{e \ni v} x_e &= 0 \quad \forall_{v \in V \setminus \{s,t\}} \\ -1 \leq x_e &\leq 1 \quad \forall_{e \in E} \end{aligned} \tag{12.1}$$

where for every vertex v, summing over $e \ni v$ means summing over all the edges that touch v.

The maximum flow problem can be thought of as the task of maximizing $\sum_{e \ni s} x_e$ over all the vectors $x \in \mathbb{R}^m$ that satisfy the above conditions (12.1). Maximizing a linear function $\ell(x)$ over the set of $x \in \mathbb{R}^m$ that satisfy certain linear equalities and inequalities is known as *linear programming*. Luckily, there are polynomial-time algorithms for solving linear programming, and hence we can solve the maximum flow (and so, equivalently, minimum cut) problem in polynomial time. In fact, there are much better algorithms for maximum-flow/minimum-cut, even for weighted directed graphs, with currently the record standing at $O(\min\{m^{10/7}, m\sqrt{n}\})$ time.

Solved Exercise 12.1 — Global minimum cut. Given a graph $G = (V, E)$, define the *global* minimum cut of G to be the minimum over all $S \subseteq V$ with $S \neq \emptyset$ and $S \neq V$ of the number of edges cut by S. Prove that there is a polynomial-time algorithm to compute the global minimum cut of a graph.

■

Solution:

By the above we know that there is a polynomial-time algorithm A that on input (G, s, t) finds the minimum s, t cut in the graph

G. Using A, we can obtain an algorithm B that on input a graph G computes the global minimum cut as follows:

1. For every distinct pair $s, t \in V$, Algorithms B sets $k_{s,t} \leftarrow A(G, s, t)$.
2. B returns the minimum of $k_{s,t}$ over all distinct pairs s, t

The running time of B will be $O(n^2)$ times the running time of A and hence polynomial time. Moreover, if the global minimum cut is S, then when B reaches an iteration with $s \in S$ and $t \notin S$ it will obtain the value of this cut, and hence the value output by B will be the value of the global minimum cut.

The above is our first example of a *reduction* in the context of polynomial-time algorithms. Namely, we reduced the task of computing the global minimum cut to the task of computing minimum s, t cuts.

■

12.1.5 Finding the maximum cut in a graph

The *maximum cut* problem is the task of finding, given an input graph $G = (V, E)$, the subset $S \subseteq V$ that *maximizes* the number of edges cut by S. (We can also define an s, t-cut variant of the maximum cut like we did for minimum cut; the two variants have similar complexity but the global maximum cut is more common in the literature.) Like its cousin the minimum cut problem, the maximum cut problem is also very well motivated. For example, maximum cut arises in VLSI design, and also has some surprising relation to analyzing the Ising model in statistical physics.

Surprisingly, while (as we've seen) there is a polynomial-time algorithm for the minimum cut problem, there is no known algorithm solving *maximum cut* much faster than the trivial "brute force" algorithm that tries all 2^n possibilities for the set S.

12.1.6 A note on convexity

There is an underlying reason for the sometimes radical difference between the difficulty of maximizing and minimizing a function over a domain. If $D \subseteq \mathbb{R}^n$, then a function $f : D \to R$ is *convex* if for every $x, y \in D$ and $p \in [0,1]$ $f(px + (1-p)y) \leq pf(x) + (1-p)f(y)$. That is, f applied to the p-weighted midpoint between x and y is smaller than the p-weighted average value of f. If D itself is convex (which means that if x, y are in D then so is the line segment between them), then this means that if x is a *local minimum* of f then it is also a *global minimum*. The reason is that if $f(y) < f(x)$ then every point $z = px + (1-p)y$ on the line segment between x and y will satisfy $f(z) \leq pf(x) + (1-p)f(y) < f(x)$ and hence in particular x cannot

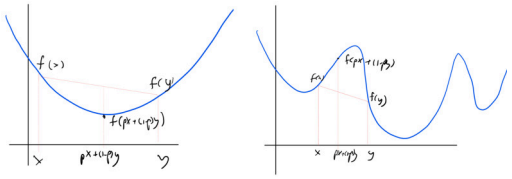


Figure 12.4: In a *convex* function f (left figure), for every x and y and $p \in [0,1]$ it holds that $f(px + (1-p)y) \leq p \cdot f(x) + (1-p) \cdot f(y)$. In particular this means that every *local minimum* of f is also a *global minimum*. In contrast in a *non-convex* function there can be many local minima.

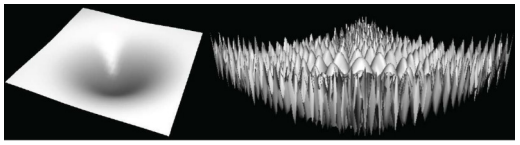


Figure 12.5: In the high dimensional case, if f is a *convex* function (left figure) the global minimum is the only local minimum, and we can find it by a local-search algorithm which can be thought of as dropping a marble and letting it "slide down" until it reaches the global minimum. In contrast, a non-convex function (right figure) might have an exponential number of local minima in which any local-search algorithm could get stuck.

be a local minimum. Intuitively, local minima of functions are much easier to find than global ones: after all, any "local search" algorithm that keeps finding a nearby point on which the value is lower, will eventually arrive at a local minimum. One example of such a local search algorithm is gradient descent which takes a sequence of small steps, each one in the direction that would reduce the value by the most amount based on the current derivative.

Indeed, under certain technical conditions, we can often efficiently find the minimum of convex functions over a convex domain, and this is the reason why problems such as minimum cut and shortest path are easy to solve. On the other hand, *maximizing* a convex function over a convex domain (or equivalently, minimizing a *concave* function) can often be a hard computational task. A *linear* function is both convex and concave, which is the reason that both the maximization and minimization problems for linear functions can be done efficiently.

The minimum cut problem is not a priori a convex minimization task, because the set of potential cuts is *discrete* and not continuous. However, it turns out that we can embed it in a continuous and convex set via the (linear) maximum flow problem. The "max flow min cut" theorem ensures that this embedding is "tight" in the sense that the minimum "fractional cut" that we obtain through the maximum-flow linear program will be the same as the true minimum cut. Unfortunately, we don't know of such a tight embedding in the setting of the *maximum* cut problem.

Convexity arises time and again in the context of efficient computation. For example, one of the basic tasks in machine learning is *empirical risk minimization*. This is the task of finding a classifier for a given set of *training examples*. That is, the input is a list of labeled examples $(x_0, y_0), \ldots, (x_{m-1}, y_{m-1})$, where each $x_i \in \{0,1\}^n$ and $y_i \in \{0,1\}$, and the goal is to find a *classifier* $h : \{0,1\}^n \to \{0,1\}$ (or sometimes $h : \{0,1\}^n \to \mathbb{R}$) that minimizes the number of *errors*. More generally, we want to find h that minimizes

$$\sum_{i=0}^{m-1} L(y_i, h(x_i))$$

where L is some *loss function* measuring how far is the predicted label $h(x_i)$ from the true label y_i. When L is the *square loss* function $L(y, y') = (y - y')^2$ and h is a *linear function*, empirical risk minimization corresponds to the well-known convex minimization task of *linear regression*. In other cases, when the task is *non-convex*, there can be many global or local minima. That said, even if we don't find the global (or even a local) minima, this continuous embedding can still help us. In particular, when running a local improvement algorithm

such as Gradient Descent, we might still find a function h that is "useful" in the sense of having a small error on future examples from the same distribution.

12.2 BEYOND GRAPHS

Not all computational problems arise from graphs. We now list some other examples of computational problems that are of great interest.

12.2.1 SAT

A *propositional formula* φ involves n variables $x_1, \ldots, x_n$ and the logical operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$, also denoted as $\bar{\cdot}$). We say that such a formula is in *conjunctive normal form* (CNF for short) if it is an AND of ORs of variables or their negations (we call a term of the form x_i or $\overline{x}_i$ a *literal*). For example, this is a CNF formula

$$(x_7 \vee \overline{x}_{22} \vee x_{15}) \wedge (x_{37} \vee x_{22}) \wedge (x_{55} \vee \overline{x}_7)$$

The *satisfiability problem* is the task of determining, given a CNF formula φ, whether or not there exists a *satisfying assignment* for φ. A satisfying assignment for φ is a string $x \in \{0,1\}^n$ such that φ evaluates to *True* if we assign its variables the values of x. The SAT problem might seem as an abstract question of interest only in logic but in fact SAT is of huge interest in industrial optimization, with applications including manufacturing planning, circuit synthesis, software verification, air-traffic control, scheduling sports tournaments, and more.

2SAT. We say that a formula is a k-CNF it is an AND of ORs where each OR involves exactly k literals. The k-SAT problem is the restriction of the satisfiability problem for the case that the input formula is a k-CNF. In particular, the *2SAT problem* is to find out, given a 2-CNF formula φ, whether there is an assignment $x \in \{0,1\}^n$ that *satisfies* φ, in the sense that it makes it evaluate to 1 or "True". The trivial, brute-force, algorithm for 2SAT will enumerate all the 2^n assignments $x \in \{0,1\}^n$ but fortunately we can do much better. The key is that we can think of every constraint of the form $\ell_i \vee \ell_j$ (where ℓ_i, ℓ_j are *literals*, corresponding to variables or their negations) as an *implication* $\overline{\ell}_i \Rightarrow \ell_j$, since it corresponds to the constraints that if the literal $\ell'_i = \overline{\ell}_i$ is true then it must be the case that ℓ_j is true as well. Hence we can think of φ as a directed graph between the $2n$ literals, with an edge from ℓ_i to ℓ_j corresponding to an implication from the former to the latter. It can be shown that φ is unsatisfiable if and only if there is a variable x_i such that there is a directed path from x_i to $\overline{x}_i$ as well as a directed path from $\overline{x}_i$ to x_i (see Exercise 12.2). This reduces 2SAT to the (efficiently solvable) problem of determining connectivity in directed graphs.

3SAT. The 3SAT problem is the task of determining satisfiability for 3CNFs. One might think that changing from two to three would not make that much of a difference for complexity. One would be wrong. Despite much effort, we do not know of a significantly better than brute force algorithm for 3SAT (the best known algorithms take roughly 1.3^n steps).

Interestingly, a similar issue arises time and again in computation, where the difference between two and three often corresponds to the difference between tractable and intractable. We do not fully understand the reasons for this phenomenon, though the notion of **NP** completeness we will see later does offer a partial explanation. It may be related to the fact that optimizing a polynomial often amounts to equations on its derivative. The derivative of a quadratic polynomial is linear, while the derivative of a cubic is quadratic, and, as we will see, the difference between solving linear and quadratic equations can be quite profound.

12.2.2 Solving linear equations

One of the most useful problems that people have been solving time and again is solving n linear equations in n variables. That is, solve equations of the form

$$\begin{array}{cccc} a_{0,0}x_0 + a_{0,1}x_1 & + \cdots & + a_{0,n-1}x_{n-1} & = b_0 \\ a_{1,0}x_0 + a_{1,1}x_1 & + \cdots & + a_{1,n-1}x_{n-1} & = b_1 \\ \vdots + \vdots & + \vdots & + \vdots & = \vdots \\ a_{n-1,0}x_0 + a_{n-1,1}x_1 & + \cdots & + a_{n-1,n-1}x_{n-1} & = b_{n-1} \end{array}$$

where $\{a_{i,j}\}_{i,j\in[n]}$ and $\{b_i\}_{i\in[n]}$ are real (or rational) numbers. More compactly, we can write this as the equations $Ax = b$ where A is an $n \times n$ matrix, and we think of x, b are column vectors in $\mathbb{R}^n$.

The standard Gaussian elimination algorithm can be used to solve such equations in polynomial time (i.e., determine if they have a solution, and if so, to find it). As we discussed above, if we are willing to allow some loss in precision, we even have algorithms that handle linear *inequalities*, also known as linear programming. In contrast, if we insist on *integer* solutions, the task of solving for linear equalities or inequalities is known as integer programming, and the best known algorithms are exponential time in the worst case.

R

Remark 12.4 — Bit complexity of numbers. Whenever we discuss problems whose inputs correspond to numbers, the input length corresponds to how many bits are needed to describe the number (or, as is equivalent up to a constant factor, the number of digits

in base 10, 16 or any other constant). The difference between the length of the input and the magnitude of the number itself can be of course quite profound. For example, most people would agree that there is a huge difference between having a billion (i.e. 10^9) dollars and having nine dollars. Similarly there is a huge difference between an algorithm that takes n steps on an n-bit number and an algorithm that takes 2^n steps.

One example is the problem (discussed below) of finding the prime factors of a given integer N. The natural algorithm is to search for such a factor by trying all numbers from 1 to N, but that would take N steps which is *exponential* in the input length, which is the number of bits needed to describe N. (The running time of this algorithm can be easily improved to roughly $\sqrt{N}$, but this is still exponential (i.e., $2^{n/2}$) in the number n of bits to describe N.) It is an important and long open question whether there is such an algorithm that runs in time polynomial in the input length (i.e., polynomial in $\log N$).

12.2.3 Solving quadratic equations

Suppose that we want to solve not just *linear* but also equations involving *quadratic* terms of the form $a_{i,j,k}x_jx_k$. That is, suppose that we are given a set of quadratic polynomials $p_1, \ldots, p_m$ and consider the equations $\{p_i(x) = 0\}$. To avoid issues with bit representations, we will always assume that the equations contain the constraints $\{x_i^2 - x_i = 0\}_{i\in[n]}$. Since only 0 and 1 satisfy the equation $a^2 - a = 0$, this assumption means that we can restrict attention to solutions in $\{0,1\}^n$. Solving quadratic equations in several variables is a classical and extremely well motivated problem. This is the generalization of the classical case of single-variable quadratic equations that generations of high school students grapple with. It also generalizes the quadratic assignment problem, introduced in the 1950's as a way to optimize assignment of economic activities. Once again, we do not know a much better algorithm for this problem than the one that enumerates over all the 2^n possibilities.

12.3 MORE ADVANCED EXAMPLES

We now list a few more examples of interesting problems that are a little more advanced but are of significant interest in areas such as physics, economics, number theory, and cryptography.

12.3.1 Determinant of a matrix

The determinant of a $n \times n$ matrix A, denoted by $\det(A)$, is an extremely important quantity in linear algebra. For example, it is known

that $\det(A) \neq 0$ if and only if A is *non-singular*, which means that it has an inverse A^{-1}, and hence we can always uniquely solve equations of the form $Ax = b$ where x and b are n-dimensional vectors. More generally, the determinant can be thought of as a quantitative measure as to what extent A is far from being singular. If the rows of A are "almost" linearly dependent (for example, if the third row is very close to being a linear combination of the first two rows) then the determinant will be small, while if they are far from it (for example, if they are are *orthogonal* to one another, then the determinant will be large). In particular, for every matrix A, the absolute value of the determinant of A is at most the product of the norms (i.e., square root of sum of squares of entries) of the rows, with equality if and only if the rows are orthogonal to one another.

The determinant can be defined in several ways. One way to define the determinant of an $n \times n$ matrix A is:

$$\det(A) = \sum_{\pi \in S_n} \text{sign}(\pi) \prod_{i \in [n]} A_{i,\pi(i)} \tag{12.2}$$

where S_n is the set of all permutations from $[n]$ to $[n]$ and the sign of a permutation π is equal to -1 raised to the power of the number of *inversions* in π (pairs i, j such that $i > j$ but $\pi(i) < \pi(j)$).

This definition suggests that computing $\det(A)$ might require summing over $|S_n|$ terms which would take exponential time since $|S_n| = n! > 2^n$. However, there are other ways to compute the determinant. For example, it is known that det is the only function that satisfies the following conditions:

1. $\det(AB) = \det(A)\det(B)$ for every square matrices A, B.
2. For every $n \times n$ *triangular* matrix T with diagonal entries $d_0, \ldots, d_{n-1}$, $\det(T) = \prod_{i=0}^{n} d_i$. In particular $\det(I) = 1$ where I is the identity matrix. (A *triangular* matrix is one in which either all entries below the diagonal, or all entries above the diagonal, are zero.)
3. $\det(S) = -1$ where S is a "swap matrix" that corresponds to swapping two rows or two columns of I. That is, there are two coordinates a, b such that for every i, j, $S_{i,j} = \begin{cases} 1 & i = j \text{ , } i \notin \{a, b\} \\ 1 & \{i, j\} = \{a, b\} \\ 0 & \text{otherwise} \end{cases}$.

Using these rules and the Gaussian elimination algorithm, it is possible to tell whether A is singular or not, and in the latter case, decompose A as a product of a polynomial number of swap matrices and triangular matrices. (Indeed one can verify that the row operations in Gaussian elimination corresponds to either multiplying by a

swap matrix or by a triangular matrix.) Hence we can compute the determinant for an $n \times n$ matrix using a polynomial time of arithmetic operations.

12.3.2 Permanent of a matrix

Given an $n \times n$ matrix A, the *permanent* of A is defined as

$$\text{perm}(A) = \sum_{\pi \in S_n} \prod_{i \in [n]} A_{i,\pi(i)} \ . \tag{12.3}$$

That is, $\text{perm}(A)$ is defined analogously to the determinant in (12.2) except that we drop the term $\text{sign}(\pi)$. The permanent of a matrix is a natural quantity, and has been studied in several contexts including combinatorics and graph theory. It also arises in physics where it can be used to describe the quantum state of multiple Boson particles (see here and here).

Permanent modulo 2. If the entries of A are integers, then we can define the *Boolean* function $perm_2$ which outputs on input a matrix A the result of the permanent of A modulo 2. It turns out that we can compute $perm_2(A)$ in polynomial time. The key is that modulo 2, $-x$ and $+x$ are the same quantity and hence, since the only difference between (12.2) and (12.3) is that some terms are multiplied by -1, $\det(A) \mod 2 = \text{perm}(A) \mod 2$ for every A.

Permanent modulo 3. Emboldened by our good fortune above, we might hope to be able to compute the permanent modulo any prime p and perhaps in full generality. Alas, we have no such luck. In a similar "two to three" type of a phenomenon, we do not know of a much better than brute force algorithm to even compute the permanent modulo 3.

12.3.3 Finding a zero-sum equilibrium

A *zero sum game* is a game between two players where the payoff for one is the same as the penalty for the other. That is, whatever the first player gains, the second player loses. As much as we want to avoid them, zero sum games do arise in life, and the one good thing about them is that at least we can compute the optimal strategy.

A zero sum game can be specified by an $n \times n$ matrix A, where if player 1 chooses action i and player 2 chooses action j then player one gets $A_{i,j}$ and player 2 loses the same amount. The famous Min Max Theorem by John von Neumann states that if we allow probabilistic or "mixed" strategies (where a player does not choose a single action but rather a *distribution* over actions) then it does not matter who plays first and the end result will be the same. Mathematically the min max theorem is that if we let Δ_n be the set of probability distributions over

$[n]$ (i.e., non-negative columns vectors in $\mathbb{R}^n$ whose entries sum to 1) then

$$\max_{p\in\Delta_n}\min_{q\in\Delta_n} p^\top Aq = \min_{q\in\Delta_n}\max_{p\in\Delta_n} p^\top Aq \tag{12.4}$$

The min-max theorem turns out to be a corollary of linear programming duality, and indeed the value of (12.4) can be computed efficiently by a linear program.

12.3.4 Finding a Nash equilibrium

Fortunately, not all real-world games are zero sum, and we do have more general games, where the payoff of one player does not necessarily equal the loss of the other. John Nash won the Nobel prize for showing that there is a notion of *equilibrium* for such games as well. In many economic texts it is taken as an article of faith that when actual agents are involved in such a game then they reach a Nash equilibrium. However, unlike zero sum games, we do not know of an efficient algorithm for finding a Nash equilibrium given the description of a general (non-zero-sum) game. In particular this means that, despite economists' intuitions, there are games for which natural strategies will take an exponential number of steps to converge to an equilibrium.

12.3.5 Primality testing

Another classical computational problem, that has been of interest since the ancient Greeks, is to determine whether a given number N is prime or composite. Clearly we can do so by trying to divide it with all the numbers in $2,\ldots,N-1$, but this would take at least N steps which is *exponential* in its bit complexity $n = \log N$. We can reduce these N steps to $\sqrt{N}$ by observing that if N is a composite of the form $N = PQ$ then either P or Q is smaller than $\sqrt{N}$. But this is still quite terrible. If N is a 1024 bit integer, $\sqrt{N}$ is about 2^{512}, and so running this algorithm on such an input would take much more than the lifetime of the universe.

Luckily, it turns out we can do radically better. In the 1970's, Rabin and Miller gave *probabilistic* algorithms to determine whether a given number N is prime or composite in time $poly(n)$ for $n = \log N$. We will discuss the probabilistic model of computation later in this course. In 2002, Agrawal, Kayal, and Saxena found a deterministic $poly(n)$ time algorithm for this problem. This is surely a development that mathematicians from Archimedes till Gauss would have found exciting.

12.3.6 Integer factoring

Given that we can efficiently determine whether a number N is prime or composite, we could expect that in the latter case we could also efficiently *find* the factorization of N. Alas, no such algorithm is known. In a surprising and exciting turn of events, the *non-existence* of such an algorithm has been used as a basis for encryptions, and indeed it underlies much of the security of the world wide web. We will return to the factoring problem later in this course. We remark that we do know much better than brute force algorithms for this problem. While the brute force algorithms would require $2^{\Omega(n)}$ time to factor an n-bit integer, there are known algorithms running in time roughly $2^{O(\sqrt{n})}$ and also algorithms that are widely believed (though not fully rigorously analyzed) to run in time roughly $2^{O(n^{1/3})}$. (By "roughly" we mean that we neglect factors that are polylogarithmic in n.)

12.4 OUR CURRENT KNOWLEDGE

The difference between an exponential and polynomial time algorithm might seem merely "quantitative" but it is in fact extremely significant. As we've already seen, the brute force exponential time algorithm runs out of steam very very fast, and as Edmonds says, in practice there might not be much difference between a problem where the best algorithm is exponential and a problem that is not solvable at all. Thus the efficient algorithms we mentioned above are widely used and power many computer science applications. Moreover, a polynomial-time algorithm often arises out of significant insight to the problem at hand, whether it is the "max-flow min-cut" result, the solvability of the determinant, or the group theoretic structure that enables primality testing. Such insight can be useful regardless of its computational implications.

At the moment we do not know whether the "hard" problems are truly hard, or whether it is merely because we haven't yet found the right algorithms for them. However, we will now see that there are problems that do *inherently require* exponential time. We just don't know if any of the examples above fall into that category.

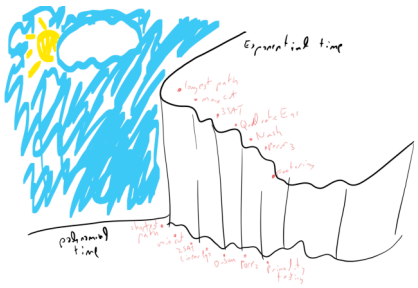


Figure 12.6: The current computational status of several interesting problems. For all of them we either know a polynomial-time algorithm or the known algorithms require at least 2^{n^c} for some $c > 0$. In fact for all except the *factoring* problem, we either know an $O(n^3)$ time algorithm or the best known algorithm require at least $2^{\Omega(n)}$ time where n is a natural parameter such that there is a brute force algorithm taking roughly 2^n or $n!$ time. Whether this "cliff" between the easy and hard problem is a real phenomenon or a reflection of our ignorance is still an open question.

Chapter Recap

- There are many natural problems that have polynomial-time algorithms, and other natural problems that we'd love to solve, but for which the best known algorithms are exponential.
- Often a polynomial time algorithm relies on discovering some hidden structure in the problem, or finding a surprising equivalent formulation for it.

- There are many interesting problems where there is an *exponential gap* between the best known algorithm and the best algorithm that we can rule out. Closing this gap is one of the main open questions of theoretical computer science.

12.5 EXERCISES

Exercise 12.1 — exponential time algorithm for longest path. The naive algorithm for computing the longest path in a given graph could take more than $n!$ steps. Give a $poly(n)2^n$ time algorithm for the longest path problem in n vertex graphs.[2]

[2] **Hint:** Use dynamic programming to compute for every $s, t \in [n]$ and $S \subseteq [n]$ the value $P(s,t,S)$ which equals 1 if there is a simple path from s to t that uses exactly the vertices in S. Do this iteratively for S's of growing sizes.

■

Exercise 12.2 — 2SAT algorithm. For every 2CNF φ, define the graph G_φ on $2n$ vertices corresponding to the literals $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$, such that there is an edge $\overrightarrow{\ell_i\ \ell_j}$ iff the constraint $\overline{\ell}_i \vee \ell_j$ is in φ. Prove that φ is unsatisfiable if and only if there is some i such that there is a path from x_i to $\overline{x}_i$ and from $\overline{x}_i$ to x_i in G_φ. Show how to use this to solve 2SAT in polynomial time.

■

Exercise 12.3 — Reductions for showing algorithms. The following fact is true: there is a polynomial-time algorithm *BIP* that on input a graph $G = (V, E)$ outputs 1 if and only if the graph is *bipartite*: there is a partition of V to disjoint parts S and T such that every edge $(u, v) \in E$ satisfies either $u \in S$ and $v \in T$ or $u \in T$ and $v \in S$. Use this fact to prove that there is a polynomial-time algorithm to compute that following function *CLIQUEPARTITION* that on input a graph $G = (V, E)$ outputs 1 if and only if there is a partition of V the graph into two parts S and T such that both S and T are *cliques*: for every pair of distinct vertices $u, v \in S$, the edge (u, v) is in E and similarly for every pair of distinct vertices $u, v \in T$, the edge (u, v) is in E.

■

12.6 BIBLIOGRAPHICAL NOTES

The classic undergraduate introduction to algorithms text is [Cor+09]. Two texts that are less "encyclopedic" are Kleinberg and Tardos [KT06], and Dasgupta, Papadimitriou and Vazirani [DPV08]. Jeff Erickson's book is an excellent algorithms text that is freely available online.

The origins of the minimum cut problem date to the Cold War. Specifically, Ford and Fulkerson discovered their max-flow/min-cut algorithm in 1955 as a way to find out the minimum amount of train

tracks that would need to be blown up to disconnect Russia from the rest of Europe. See the survey [Sch05] for more.

Some algorithms for the longest path problem are given in [Wil09; Bjo14].

12.7 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

Learning Objectives:

- Formally modeling running time, and in particular notions such as $O(n)$ or $O(n^3)$ time algorithms.
- The classes P and EXP modelling polynomial and exponential time respectively.
- The *time hierarchy theorem*, that in particular says that for every $k \geq 1$ there are functions we *can* compute in $O(n^{k+1})$ time but *can not* compute in $O(n^k)$ time.
- The class $\mathrm{P}_{/\mathrm{poly}}$ of *non-uniform* computation and the result that $\mathrm{P} \subseteq \mathrm{P}_{/\mathrm{poly}}$

13
Modeling running time

> "When the measure of the problem-size is reasonable and when the sizes assume values arbitrarily large, an asymptotic estimate of ... the order of difficulty of [an] algorithm .. is theoretically important. It cannot be rigged by making the algorithm artificially difficult for smaller sizes", Jack Edmonds, "Paths, Trees, and Flowers", 1963

> *Max Newman:* It is all very well to say that a machine could ... do this or that, but ... what about the time it would take to do it?
>
> *Alan Turing:* To my mind this time factor is the one question which will involve all the real technical difficulty.
>
> BBC radio panel on "Can automatic Calculating Machines Be Said to Think?", 1952

In Chapter 12 we saw examples of efficient algorithms, and made some claims about their running time, but did not give a mathematically precise definition for this concept. We do so in this chapter, using the models of Turing machines and RAM machines (or equivalently NAND-TM and NAND-RAM) we have seen before. The running time of an algorithm is not a fixed number since any non-trivial algorithm will take longer to run on longer inputs. Thus, what we want to measure is the *dependence* between the number of steps the algorithms takes and the length of the input. In particular we care about the distinction between algorithms that take at most *polynomial time* (i.e., $O(n^c)$ time for some constant c) and problems for which every algorithm requires at least *exponential time* (i.e., $\Omega(2^{n^c})$ for some c). As mentioned in Edmond's quote in Chapter 12, the difference between these two can sometimes be as important as the difference between being computable and uncomputable.

This chapter: A non-mathy overview

In this chapter we formally define what it means for a function to be computable in a certain number of steps. As discussed in Chapter 12, running time is not a number, rather

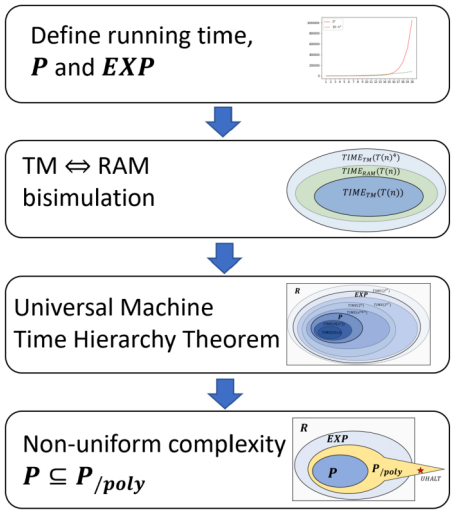


Figure 13.1: Overview of the results of this chapter.

what we care about is the *scaling behaviour* of the number of steps as the input size grows. We can use either Turing machines or RAM machines to give such a formal definition - it turns out that this doesn't make a difference at the resolution we care about. We make several important definitions and prove some important theorems in this chapter. We will define the main *time complexity classes* we use in this book, and also show the *Time Hierarchy Theorem* which states that given more resources (more time steps per input size) we can compute more functions.

To put this in more "mathy" language, in this chapter we define what it means for a function $F : \{0,1\}^* \rightarrow \{0,1\}^*$ to be *computable in time $T(n)$ steps*, where T is some function mapping the length n of the input to the number of computation steps allowed. Using this definition we will do the following (see also Fig. 13.1):

- We define the class **P** of Boolean functions that can be computed in polynomial time and the class **EXP** of functions that can be computed in exponential time. Note that **P** $\subseteq$ **EXP**. If we can compute a function in polynomial time, we can certainly compute it in exponential time.

- We show that the times to compute a function using a Turing machine and using a RAM machine (or NAND-RAM program) are *polynomially related*. In particular this means that the classes **P** and **EXP** are identical regardless of whether they are defined using Turing machines or RAM machines / NAND-RAM programs.

- We give an *efficient* universal NAND-RAM program and use this to establish the *time hierarchy theorem* that in particular implies that **P** is a *strict subset* of **EXP**.
- We relate the notions defined here to the *non-uniform* models of Boolean circuits and NAND-CIRC programs defined in Chapter 3. We define $\mathbf{P}_{/\mathbf{poly}}$ to be the class of functions that can be computed by a *sequence* of polynomial-sized circuits. We prove that $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$ and that $\mathbf{P}_{/\mathbf{poly}}$ contains *uncomputable* functions.

13.1 FORMALLY DEFINING RUNNING TIME

Our models of computation (Turing machines, NAND-TM and NAND-RAM programs and others) all operate by executing a sequence of instructions on an input one step at a time. We can define the *running time* of an algorithm M in one of these models by measuring the number of steps M takes on input x as a *function of the length* $|x|$ *of the input*. We start by defining running time with respect to Turing machines:

Definition 13.1 — Running time (Turing Machines). Let $T : \mathbb{N} \to \mathbb{N}$ be some function mapping natural numbers to natural numbers. We say that a function $F : \{0,1\}^* \to \{0,1\}^*$ is *computable in $T(n)$ Turing-Machine time* (*TM-time for short*) if there exists a Turing machine M such that for every sufficiently large n and every $x \in \{0,1\}^n$, when given input x, the machine M halts after executing at most $T(n)$ steps and outputs $F(x)$.

We define $TIME_{\mathsf{TM}}(T(n))$ to be the set of Boolean functions (functions mapping $\{0,1\}^*$ to $\{0,1\}$) that are computable in $T(n)$ TM time.

Big Idea 17 For a function $F : \{0,1\}^* \to \{0,1\}$ and $T : \mathbb{N} \to \mathbb{N}$, we can formally define what it means for F to be computable in time at most $T(n)$ where n is the size of the input.

Definition 13.1 is not very complicated but is one of the most important definitions of this book. As usual, $TIME_{\mathsf{TM}}(T(n))$ is a class of *functions*, not of *machines*. If M is a Turing machine then a statement such as "M is a member of $TIME_{\mathsf{TM}}(n^2)$" does not make sense. The concept of TM-time as defined here is sometimes known as "single-tape Turing machine time" in the literature, since some texts consider Turing machines with more than one working tape.

The relaxation of considering only "sufficiently large" n's is not very important but it is convenient since it allows us to avoid dealing explicitly with un-interesting "edge cases".

While the notion of being computable within a certain running time can be defined for every function, the class $TIME_{\mathsf{TM}}(T(n))$ is a class of *Boolean functions* that have a single bit of output. This choice is not very important, but is made for simplicity and convenience later on. In fact, every non-Boolean function has a computationally equivalent Boolean variant, see Exercise 13.3.

Solved Exercise 13.1 — Example of time bounds. Prove that $TIME_{\mathsf{TM}}(10 \cdot n^3) \subseteq TIME_{\mathsf{TM}}(2^n)$.

Solution:

The proof is illustrated in Fig. 13.2. Suppose that $F \in TIME_{\mathsf{TM}}(10 \cdot n^3)$ and hence there exist some number N_0 and a machine M such that for every $n > N_0$, and $x \in \{0,1\}^*$, $M(x)$ outputs $F(x)$ within at most $10 \cdot n^3$ steps. Since $10 \cdot n^3 = o(2^n)$, there is some number N_1 such that for every $n > N_1$, $10 \cdot n^3 < 2^n$. Hence for every $n > \max\{N_0, N_1\}$, $M(x)$ will output $F(x)$ within at most 2^n steps, demonstrating that $F \in TIME_{\mathsf{TM}}(2^n)$.

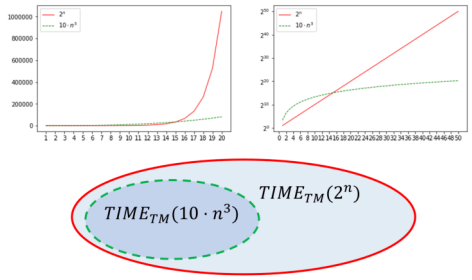


Figure 13.2: Comparing $T(n) = 10n^3$ with $T'(n) = 2^n$ (on the right figure the Y axis is in log scale). Since for every large enough n, $T'(n) \geq T(n)$, $TIME_{\mathsf{TM}}(T(n)) \subseteq TIME_{\mathsf{TM}}(T'(n))$.

13.1.1 Polynomial and Exponential Time

Unlike the notion of computability, the exact running time can be a function of the model we use. However, it turns out that if we only care about "coarse enough" resolution (as will most often be the case) then the choice of the model, whether Turing machines, RAM machines, NAND-TM/NAND-RAM programs, or C/Python programs, does not matter. This is known as the *extended* Church-Turing Thesis. Specifically we will mostly care about the difference between *polynomial* and *exponential* time.

The two main time complexity classes we will be interested in are the following:

- **Polynomial time:** A function $F : \{0,1\}^* \to \{0,1\}$ is *computable in polynomial time* if it is in the class $\mathbf{P} = \cup_{c \in \{1,2,3,\ldots\}} TIME_{\mathsf{TM}}(n^c)$. That is, $F \in \mathbf{P}$ if there is an algorithm to compute F that runs in time at most *polynomial* (i.e., at most n^c for some constant c) in the length of the input.

- **Exponential time:** A function $F : \{0,1\}^* \to \{0,1\}$ is *computable in exponential time* if it is in the class $\mathbf{EXP} = \cup_{c \in \{1,2,3,\ldots\}} TIME_{\mathsf{TM}}(2^{n^c})$. That is, $F \in \mathbf{EXP}$ if there is an algorithm to compute F that runs in

time at most *exponential* (i.e., at most 2^{n^c} for some constant c) in the length of the input.

In other words, these are defined as follows:

Definition 13.2 — P and EXP. Let $F : \{0,1\}^* \to \{0,1\}$. We say that $F \in \mathbf{P}$ if there is a polynomial $p : \mathbb{N} \to \mathbb{R}$ and a Turing machine M such that for every $x \in \{0,1\}^*$, when given input x, the Turing machine halts within at most $p(|x|)$ steps and outputs $F(x)$.

We say that $F \in \mathbf{EXP}$ if there is a polynomial $p : \mathbb{N} \to \mathbb{R}$ and a Turing machine M such that for every $x \in \{0,1\}^*$, when given input x, M halts within at most $2^{p(|x|)}$ steps and outputs $F(x)$.

Please take the time to make sure you understand these definitions. In particular, sometimes students think of the class **EXP** as corresponding to functions that are *not* in **P**. However, this is not the case. If F is in **EXP** then it *can* be computed in exponential time. This does not mean that it cannot be computed in polynomial time as well.

Solved Exercise 13.2 — Differerent definitions of P. Prove that **P** as defined in Definition 13.2 is equal to $\cup_{c\in\{1,2,3,\ldots\}} TIME_{\mathsf{TM}}(n^c)$

Solution:

To show these two sets are equal we need to show that $\mathbf{P} \subseteq \cup_{c\in\{1,2,3,\ldots\}} TIME_{\mathsf{TM}}(n^c)$ and $\cup_{c\in\{1,2,3,\ldots\}} TIME_{\mathsf{TM}}(n^c) \subseteq \mathbf{P}$. We start with the former inclusion. Suppose that $F \in \mathbf{P}$. Then there is some polynomial $p : \mathbb{N} \to \mathbb{R}$ and a Turing machine M such that M computes F and M halts on every input x within at most $p(|x|)$ steps. We can write the polynomial $p : \mathbb{N} \to \mathbb{R}$ in the form $p(n) = \sum_{i=0}^{d} a_i n^i$ where $a_0, \ldots, a_d \in \mathbb{R}$, and we assume that a_d is non-zero (or otherwise we just let d correspond to the largest number such that a_d is non-zero). The *degree* of p is the number d. Since $n^d = o(n^{d+1})$, no matter what the coefficient a_d is, for large enough n, $p(n) < n^{d+1}$ which means that the Turing machine M will halt on inputs of length n within fewer than n^{d+1} steps, and hence $F \in TIME_{\mathsf{TM}}(n^{d+1}) \subseteq \cup_{c\in\{1,2,3,\ldots\}} TIME_{\mathsf{TM}}(n^c)$.

For the second inclusion, suppose that $F \in \cup_{c\in\{1,2,3,\ldots\}} TIME_{\mathsf{TM}}(n^c)$. Then there is some positive $c \in \mathbb{N}$ such that $F \in TIME_{\mathsf{TM}}(n^c)$ which means that there is a Turing machine M and some number N_0 such that M computes F and for every $n > N_0$, M halts on length n

inputs within at most n^c steps. Let T_0 be the maximum number of steps that M takes on inputs of length at most N_0. Then if we define the polynomial $p(n) = n^c + T_0$ then we see that M halts on every input x within at most $p(|x|)$ steps and hence the existence of M demonstrates that $F \in \mathbf{P}$.

Since exponential time is much larger than polynomial time, $\mathbf{P} \subseteq \mathbf{EXP}$. All of the problems we listed in Chapter 12 are in **EXP**, but as we've seen, for some of them there are much better algorithms that demonstrate that they are in fact in the smaller class **P**.

P	**EXP** (but not known to be in **P**)
Shortest path	Longest Path
Min cut	Max cut
2SAT	3SAT
Linear eqs	Quad. eqs
Zerosum	Nash
Determinant	Permanent
Primality	Factoring

Table : A table of the examples from Chapter 12. All these problems are in **EXP** but only the ones on the left column are currently known to be in **P** as well (i.e., they have a polynomial-time algorithm). See also Fig. 13.3.

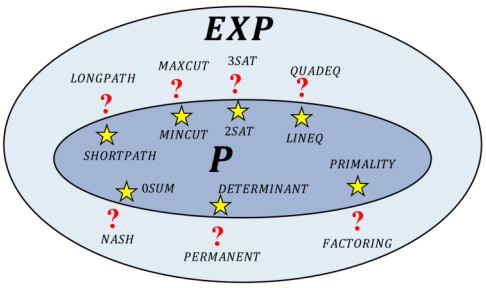


Figure 13.3: Some examples of problems that are known to be in **P** and problems that are known to be in **EXP** but not known whether or not they are in **P**. Since both **P** and **EXP** are classes of Boolean functions, in this figure we always refer to the *Boolean* (i.e., Yes/No) variant of the problems.

R

Remark 13.3 — Boolean versions of problems. Many of the problems defined in Chapter 12 correspond to *non-Boolean* functions (functions with more than one bit of output) while **P** and **EXP** are sets of Boolean functions. However, for every non-Boolean function F we can always define a computationally-equivalent Boolean function G by letting $G(x, i)$ be the i-th bit of $F(x)$ (see Exercise 13.3). Hence the table above, as well as Fig. 13.3, refer to the computationally-equivalent Boolean variants of these problems.

13.2 MODELING RUNNING TIME USING RAM MACHINES / NAND-RAM

Turing machines are a clean theoretical model of computation, but do not closely correspond to real-world computing architectures. The discrepancy between Turing machines and actual computers does not matter much when we consider the question of which functions are *computable*, but can make a difference in the context of *efficiency*.

Even a basic staple of undergraduate algorithms such as "merge sort" cannot be implemented on a Turing machine in $O(n \log n)$ time (see Section 13.8). *RAM machines* (or equivalently, NAND-RAM programs) match more closely actual computing architecture and what we mean when we say $O(n)$ or $O(n \log n)$ algorithms in algorithms courses or whiteboard coding interviews. We can define running time with respect to NAND-RAM programs just as we did for Turing machines.

Definition 13.4 — Running time (RAM). Let $T : \mathbb{N} \to \mathbb{N}$ be some function mapping natural numbers to natural numbers. We say that a function $F : \{0,1\}^* \to \{0,1\}^*$ is *computable in $T(n)$ RAM time* (*RAM-time for short*) if there exists a NAND-RAM program P such that for every sufficiently large n and every $x \in \{0,1\}^n$, when given input x, the program P halts after executing at most $T(n)$ lines and outputs $F(x)$.

We define $TIME_{\mathsf{RAM}}(T(n))$ to be the set of Boolean functions (functions mapping $\{0,1\}^*$ to $\{0,1\}$) that are computable in $T(n)$ RAM time.

Because NAND-RAM programs correspond more closely to our natural notions of running time, we will use NAND-RAM as our "default" model of running time, and hence use $TIME(T(n))$ (without any subscript) to denote $TIME_{\mathsf{RAM}}(T(n))$. However, it turns out that as long as we only care about the difference between exponential and polynomial time, this does not make much difference. The reason is that Turing machines can simulate NAND-RAM programs with at most a polynomial overhead (see also Fig. 13.4):

Theorem 13.5 — Relating RAM and Turing machines. Let $T : \mathbb{N} \to \mathbb{N}$ be a function such that $T(n) \geq n$ for every n and the map $n \mapsto T(n)$ can be computed by a Turing machine in time $O(T(n))$. Then

$$TIME_{\mathsf{TM}}(T(n)) \subseteq TIME_{\mathsf{RAM}}(10 \cdot T(n)) \subseteq TIME_{\mathsf{TM}}(T(n)^4) \; . \quad (13.1)$$

P

The technical details of Theorem 13.5, such as the condition that $n \mapsto T(n)$ is computable in $O(T(n))$ time or the constants 10 and 4 in (13.1) (which are not tight and can be improved), are not very important. In particular, all non-pathological time bound functions we encounter in practice such as $T(n) = n$, $T(n) = n \log n$, $T(n) = 2^n$ etc. will satisfy the conditions of Theorem 13.5, see also Remark 13.6.

The main message of the theorem is Turing machines and RAM machines are "roughly equivalent" in the sense that one can simulate the other with polyno-

mial overhead. Similarly, while the proof involves some technical details, it's not very deep or hard, and merely follows the simulation of RAM machines with Turing machines we saw in Theorem 8.1 with more careful "book keeping".

For example, by instantiating Theorem 13.5 with $T(n) = n^a$ and using the fact that $10n^a = o(n^{a+1})$, we see that $TIME_{\mathsf{TM}}(n^a) \subseteq TIME_{\mathsf{RAM}}(n^{a+1}) \subseteq TIME_{\mathsf{TM}}(n^{4a+4})$ which means that (by Solved Exercise 13.2)

$$\mathbf{P} = \cup_{a=1,2,\ldots} TIME_{\mathsf{TM}}(n^a) = \cup_{a=1,2,\ldots} TIME_{\mathsf{RAM}}(n^a) \ .$$

That is, we could have equally well defined **P** as the class of functions computable by *NAND-RAM programs* (instead of Turing machines) that run in time polynomial in the length of the input. Similarly, by instantiating Theorem 13.5 with $T(n) = 2^{n^a}$ we see that the class **EXP** can also be defined as the set of functions computable by NAND-RAM programs in time at most $2^{p(n)}$ where p is some polynomial. Similar equivalence results are known for many models including cellular automata, C/Python/Javascript programs, parallel computers, and a great many other models, which justifies the choice of **P** as capturing a technology-independent notion of tractability. (See Section 13.3 for more discussion of this issue.) This equivalence between Turing machines and NAND-RAM (as well as other models) allows us to pick our favorite model depending on the task at hand (i.e., "have our cake and eat it too") even when we study questions of efficiency, as long as we only care about the gap between *polynomial* and *exponential* time. When we want to *design* an algorithm, we can use the extra power and convenience afforded by NAND-RAM. When we want to *analyze* a program or prove a *negative result*, we can restrict our attention to Turing machines.

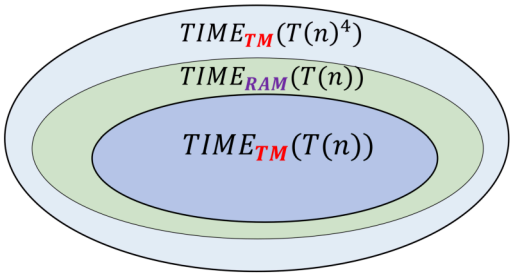


Figure 13.4: The proof of Theorem 13.5 shows that we can simulate T steps of a Turing machine with T steps of a NAND-RAM program, and can simulate T steps of a NAND-RAM program with $o(T^4)$ steps of a Turing machine. Hence $TIME_{\mathsf{TM}}(T(n)) \subseteq TIME_{\mathsf{RAM}}(10 \cdot T(n)) \subseteq TIME_{\mathsf{TM}}(T(n)^4)$.

Big Idea 18 All "reasonable" computational models are equivalent if we only care about the distinction between polynomial and exponential.

The adjective "reasonable" above refers to all scalable computational models that have been implemented, with the possible exception of *quantum computers*, see Section 13.3 and Chapter 23.

Proof Idea:

The direction $TIME_{\mathsf{TM}}(T(n)) \subseteq TIME_{\mathsf{RAM}}(10 \cdot T(n))$ is not hard to show, since a NAND-RAM program P can simulate a Turing machine M with constant overhead by storing the transition table of M in

an array (as is done in the proof of Theorem 9.1). Simulating every step of the Turing machine can be done in a constant number c of steps of RAM, and it can be shown this constant c is smaller than 10. Thus the heart of the theorem is to prove that $TIME_{\mathsf{RAM}}(T(n)) \subseteq TIME_{\mathsf{TM}}(T(n)^4)$. This proof closely follows the proof of Theorem 8.1, where we have shown that every function F that is computable by a NAND-RAM program P is computable by a Turing machine (or equivalently a NAND-TM program) M. To prove Theorem 13.5, we follow the exact same proof but just check that the overhead of the simulation of P by M is polynomial. The proof has many details, but is not deep. It is therefore much more important that you understand the *statement* of this theorem than its proof.

⋆

Proof of Theorem 13.5. We only focus on the non-trivial direction $TIME_{\mathsf{RAM}}(T(n)) \subseteq TIME_{\mathsf{TM}}(T(n)^4)$. Let $F \in TIME_{\mathsf{RAM}}(T(n))$. F can be computed in time $T(n)$ by some NAND-RAM program P and we need to show that it can also be computed in time $T(n)^4$ by a Turing machine M. This will follow from showing that F can be computed in time $T(n)^4$ by a NAND-TM program, since for every NAND-TM program Q there is a Turing machine M simulating it such that each iteration of Q corresponds to a single step of M.

As mentioned above, we follow the proof of Theorem 8.1 (simulation of NAND-RAM programs using NAND-TM programs) and use the exact same simulation, but with a more careful accounting of the number of steps that the simulation costs. Recall, that the simulation of NAND-RAM works by "peeling off" features of NAND-RAM one by one, until we are left with NAND-TM.

We will not provide the full details but will present the main ideas used in showing that every feature of NAND-RAM can be simulated by NAND-TM with at most a polynomial overhead:

1. Recall that every NAND-RAM variable or array element can contain an integer between 0 and T where T is the number of lines that have been executed so far. Therefore if P is a NAND-RAM program that computes F in $T(n)$ time, then on inputs of length n, all integers used by P are of magnitude at most $T(n)$. This means that the largest value `i` can ever reach is at most $T(n)$ and so each one of P's variables can be thought of as an array of at most $T(n)$ indices, each of which holds a natural number of magnitude at most $T(n)$. We let $\ell = \lceil \log T(n) \rceil$ be the number of bits needed to encode such numbers. (We can start off the simulation by computing $T(n)$ and ℓ.)

2. We can encode a NAND-RAM array of length $\leq T(n)$ containing numbers in $\{0, \ldots, T(n)-1\}$ as an Boolean (i.e., NAND-TM) array of $T(n)\ell = O(T(n)\log T(n))$ bits, which we can also think of as a *two dimensional array* as we did in the proof of Theorem 8.1. We encode a NAND-RAM scalar containing a number in $\{0, \ldots, T(n)-1\}$ simply by a shorter NAND-TM array of ℓ bits.

3. We can simulate the two dimensional arrays using one-dimensional arrays of length $T(n)\ell = O(T(n)\log T(n))$. All the arithmetic operations on integers use the grade-school algorithms, that take time that is polynomial in the number ℓ of bits of the integers, which is $poly(\log T(n))$ in our case. Hence we can simulate $T(n)$ steps of NAND-RAM with $O(T(n)poly(\log T(n))$ steps of a model that uses random access memory but only *Boolean-valued* one-dimensional arrays.

4. The most expensive step is to translate from random access memory to the sequential memory model of NAND-TM/Turing machines. As we did in the proof of Theorem 8.1 (see Section 8.2), we can simulate accessing an array `Foo` at some location encoded in an array `Bar` by:

   a. Copying `Bar` to some temporary array `Temp`
   b. Having an array `Index` which is initially all zeros except 1 at the first location.
   c. Repeating the following until `Temp` encodes the number 0: (*Number of repetitions is at most* $T(n)$.)
      - Decrease the number encoded temp by 1. (*Takes number of steps polynomial in* $\ell = \lceil \log T(n) \rceil$.)
      - Decrease `i` until it is equal to 0. (*Takes* $O(T(n))$ *steps.*)
      - Scan `Index` until we reach the point in which it equals 1 and then change this 1 to 0 and go one step further and write 1 in this location. (*Takes* $O(T(n))$ *steps.*)
   d. When we are done we know that if we scan `Index` until we reach the point in which `Index[i]`$= 1$ then `i` contains the value that was encoded by `Bar` (*Takes* $O(T(n))$ *steps.*)

The total cost for each such operation is $O(T(n)^2 + T(n)poly(\log T(n))) = O(T(n)^2)$ steps.

In sum, we simulate a single step of NAND-RAM using $O(T(n)^2 poly(\log T(n)))$ steps of NAND-TM, and hence the total simulation time is $O(T(n)^3 poly(\log T(n)))$ which is smaller than $T(n)^4$ for sufficiently large n.

■

R

Remark 13.6 — Nice time bounds. When considering general time bounds such we need to make sure to rule out some "pathological" cases such as functions T that don't give enough time for the algorithm to read the input, or functions where the time bound itself is uncomputable. We say that a function $T : \mathbb{N} \to \mathbb{N}$ is a *nice time bound function* (or nice function for short) if for every $n \in \mathbb{N}$, $T(n) \geq n$ (i.e., T allows enough time to read the input), for every $n' \geq n$, $T(n') \geq T(n)$ (i.e., T allows more time on longer inputs), and the map $F(x) = 1^{T(|x|)}$ (i.e., mapping a string of length n to a sequence of $T(n)$ ones) can be computed by a NAND-RAM program in $O(T(n))$ time.

All the "normal" time complexity bounds we encounter in applications such as $T(n) = 100n$, $T(n) = n^2 \log n$,$T(n) = 2^{\sqrt{n}}$, etc. are "nice". Hence from now on we will only care about the class $TIME(T(n))$ when T is a "nice" function. The computability condition is in particular typically easily satisfied. For example, for arithmetic functions such as $T(n) = n^3$, we can typically compute the binary representation of $T(n)$ in time polynomial *in the number of bits* of $T(n)$ and hence poly-logarithmic in $T(n)$. Hence the time to write the string $1^{T(n)}$ in such cases will be $T(n) + poly(\log T(n)) = O(T(n))$.

13.3 EXTENDED CHURCH-TURING THESIS (DISCUSSION)

Theorem 13.5 shows that the computational models of *Turing machines* and *RAM machines* / *NAND-RAM programs* are equivalent up to polynomial factors in the running time. Other examples of polynomially equivalent models include:

- All standard programming languages, including C/Python/-JavaScript/Lisp/etc.
- The λ calculus (see also Section 13.8).
- Cellular automata
- Parallel computers
- Biological computing devices such as DNA-based computers.

The *Extended Church Turing Thesis* is the statement that this is true for all physically realizable computing models. In other words, the extended Church Turing thesis says that for every *scalable computing device* C (which has a finite description but can be in principle used to run computation on arbitrarily large inputs), there is some constant a such that for every function $F : \{0,1\}^* \to \{0,1\}$ that C can

compute on n length inputs using an $S(n)$ amount of physical resources, F is in $TIME(S(n)^a)$. This is a strengthening of the ("plain") Church-Turing Thesis, discussed in Section 8.8, which states that the set of computable functions is the same for all physically realizable models, but without requiring the overhead in the simulation between different models to be at most polynomial.

All the current constructions of scalable computational models and programming languages conform to the Extended Church-Turing Thesis, in the sense that they can be simulated with polynomial overhead by Turing machines (and hence also by NAND-TM or NAND-RAM programs). Consequently, the classes **P** and **EXP** are robust to the choice of model, and we can use the programming language of our choice, or high level descriptions of an algorithm, to determine whether or not a problem is in **P**.

Like the Church-Turing thesis itself, the extended Church-Turing thesis is in the asymptotic setting and does not directly yield an experimentally testable prediction. However, it can be instantiated with more concrete bounds on the overhead, yielding experimentally-testable predictions such as the *Physical Extended Church-Turing Thesis* we mentioned in Section 5.6.

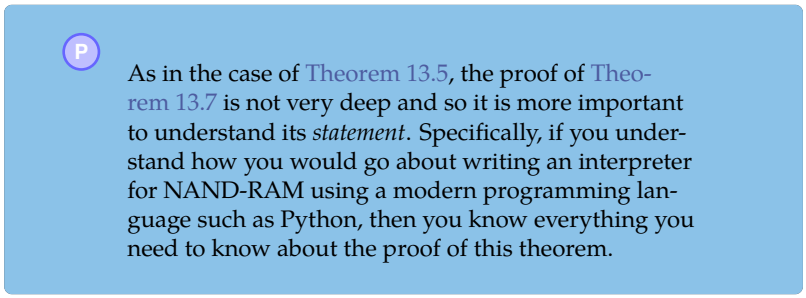
In the last hundred+ years of studying and mechanizing computation, no one has yet constructed a scalable computing device that violates the extended Church Turing Thesis. However, *quantum computing*, if realized, will pose a serious challenge to the extended Church-Turing Thesis (see Chapter 23). However, even if the promises of quantum computing are fully realized, the extended Church-Turing thesis is "morally" correct, in the sense that, while we do need to adapt the thesis to account for the possibility of quantum computing, its broad outline remains unchanged. We are still able to model computation mathematically, we can still treat programs as strings and have a universal program, we still have time hierarchy and uncomputability results, and there is still no reason to doubt the ("plain") Church-Turing thesis. Moreover, the prospect of quantum computing does not seem to make a difference for the time complexity of many (though not all!) of the concrete problems that we care about. In particular, as far as we know, out of all the example problems mentioned in Chapter 12 the complexity of only one— integer factoring— is affected by modifying our model to include quantum computers as well.

13.4 EFFICIENT UNIVERSAL MACHINE: A NAND-RAM INTERPRETER IN NAND-RAM

We have seen in Theorem 9.1 the "universal Turing machine". Examining that proof, and combining it with Theorem 13.5 , we can see that the program U has a *polynomial* overhead, in the sense that it can simulate T steps of a given NAND-TM (or NAND-RAM) program P on an input x in $O(T^4)$ steps. But in fact, by directly simulating NAND-RAM programs we can do better with only a *constant* multiplicative overhead. That is, there is a *universal NAND-RAM program* U such that for every NAND-RAM program P, U simulates T steps of P using only $O(T)$ steps. (The implicit constant in the O notation can depend on the program P but does *not* depend on the length of the input.)

Theorem 13.7 — Efficient universality of NAND-RAM. There exists a NAND-RAM program U satisfying the following:

1. (U *is a universal NAND-RAM program.*) For every NAND-RAM program P and input x, $U(P,x) = P(x)$ where by $U(P,x)$ we denote the output of U on a string encoding the pair (P,x).
2. (U *is efficient.*) There are some constants a, b such that for every NAND-RAM program P, if P halts on input x after at most T steps, then $U(P,x)$ halts after at most $C \cdot T$ steps where $C \leq a|P|^b$.



As in the case of Theorem 13.5, the proof of Theorem 13.7 is not very deep and so it is more important to understand its *statement*. Specifically, if you understand how you would go about writing an interpreter for NAND-RAM using a modern programming language such as Python, then you know everything you need to know about the proof of this theorem.

Proof of Theorem 13.7. To present a universal NAND-RAM program in full we would need to describe a precise representation scheme, as well as the full NAND-RAM instructions for the program. While this can be done, it is more important to focus on the main ideas, and so we just sketch the proof here. A specification of NAND-RAM is given in the appendix, and for the purposes of this simulation, we can simply use the representation of the NAND-RAM code as an ASCII string.

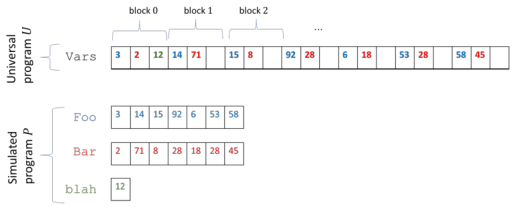


Figure 13.5: The universal NAND-RAM program U simulates an input NAND-RAM program P by storing all of P's variables inside a single array `Vars` of U. If P has t variables, then the array `Vars` is divided into blocks of length t, where the j-th coordinate of the i-th block contains the i-th element of the j-th array of P. If the j-th variable of P is scalar, then we just store its value in the zeroth block of `Vars`.

The program U gets as input a NAND-RAM program P and an input x and simulates P one step at a time. To do so, U does the following:

1. U maintains variables `program_counter`, and `number_steps` for the current line to be executed and the number of steps executed so far.

2. U initially scans the code of P to find the number t of unique variable names that P uses. It will translate each variable name into a number between 0 and $t-1$ and use an array `Program` to store P's code where for every line ℓ, `Program[`ℓ`]` will store the ℓ-th line of P where the variable names have been translated to numbers. (More concretely, we will use a constant number of arrays to separately encode the operation used in this line, and the variable names and indices of the operands.)

3. U maintains a single array `Vars` that contains all the values of P's variables. We divide `Vars` into blocks of length t. If s is a number corresponding to an array variable `Foo` of P, then we store `Foo[0]` in `Vars[`s`]`, we store `Foo[1]` in `Var_values[`$t+s$`]`, `Foo[2]` in `Vars[`$2t+s$`]` and so on and so forth (see Fig. 13.5). Generally, if the s-th variable of P is a scalar variable, then its value will be stored in location `Vars[`s`]`. If it is an array variable then the value of its i-th element will be stored in location `Vars[`$t \cdot i + s$`]`.

4. To simulate a single step of P, the program U recovers from `Program` the line corresponding to `program_counter` and executes it. Since NAND-RAM has a constant number of arithmetic operations, we can implement the logic of which operation to execute using a sequence of a constant number of if-then-else's. Retrieving from `Vars` the values of the operands of each instruction can be done using a constant number of arithmetic operations.

The setup stages take only a constant (depending on $|P|$ but not on the input x) number of steps. Once we are done with the setup, to simulate a single step of P, we just need to retrieve the corresponding line and do a constant number of "if elses" and accesses to `Vars` to simulate it. Hence the total running time to simulate T steps of the program P is at most $O(T)$ when suppressing constants that depend on the program P.

■

13.4.1 Timed Universal Turing Machine

One corollary of the efficient universal machine is the following. Given any Turing machine M, input x, and "step budget" T, we can simulate the execution of M for T steps in time that is polynomial in

T. Formally, we define a function *TIMEDEVAL* that takes the three parameters M, x, and the time budget, and outputs $M(x)$ if M halts within at most T steps, and outputs 0 otherwise. The timed universal Turing machine computes *TIMEDEVAL* in polynomial time (see Fig. 13.6). (Since we measure time as a function of the input length, we define *TIMEDEVAL* as taking the input T represented in *unary*: a string of T ones.)

Theorem 13.8 — Timed Universal Turing Machine. Let $TIMEDEVAL : \{0,1\}^* \to \{0,1\}^*$ be the function defined as

$$TIMEDEVAL(M,x,1^T) = \begin{cases} M(x) & M \text{ halts within } \leq T \text{ steps on } x \\ 0 & \text{otherwise} \end{cases}.$$

Then $TIMEDEVAL \in \mathbf{P}$.

Proof. We only sketch the proof since the result follows fairly directly from Theorem 13.5 and Theorem 13.7. By Theorem 13.5 to show that $TIMEDEVAL \in \mathbf{P}$, it suffices to give a polynomial-time NAND-RAM program to compute *TIMEDEVAL*.

Such a program can be obtained as follows. Given a Turing machine M, by Theorem 13.5 we can transform it in time polynomial in its description into a functionally-equivalent NAND-RAM program P such that the execution of M on T steps can be simulated by the execution of P on $c \cdot T$ steps. We can then run the universal NAND-RAM machine of Theorem 13.7 to simulate P for $c \cdot T$ steps, using $O(T)$ time, and output 0 if the execution did not halt within this budget. This shows that *TIMEDEVAL* can be computed by a NAND-RAM program in time polynomial in $|M|$ and linear in T, which means $TIMEDEVAL \in \mathbf{P}$.

■

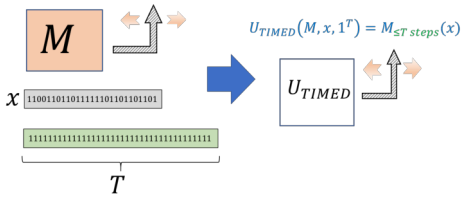


Figure 13.6: The *timed* universal Turing machine takes as input a Turing machine M, an input x, and a time bound T, and outputs $M(x)$ if M halts within at most T steps. Theorem 13.8 states that there is such a machine that runs in time polynomial in T.

13.5 THE TIME HIERARCHY THEOREM

Some functions are *uncomputable*, but are there functions that can be computed, but only at an exorbitant cost? For example, is there a function that *can* be computed in time 2^n, but *can not* be computed in time $2^{0.9n}$? It turns out that the answer is **Yes**:

Theorem 13.9 — Time Hierarchy Theorem. For every nice function $T : \mathbb{N} \to \mathbb{N}$, there is a function $F : \{0,1\}^* \to \{0,1\}$ in $TIME(T(n)\log n) \setminus TIME(T(n))$.

There is nothing special about $\log n$, and we could have used any other efficiently computable function that tends to infinity with n.

Big Idea 19 If we have more time, we can compute more functions.

R

Remark 13.10 — Simpler corollary of the time hierarchy theorem. The generality of the time hierarchy theorem can make its proof a little hard to read. It might be easier to follow the proof if you first try to prove by yourself the easier statement $\mathbf{P} \subsetneq \mathbf{EXP}$.

You can do so by showing that the following function $F : \{0,1\}^* \to \{0,1\}$ is in $\mathbf{EXP} \setminus \mathbf{P}$: for every Turing machine M and input x, $F(M,x) = 1$ if and only if M halts on x within at most $|x|^{\log|x|}$ steps. One can show that $F \in TIME(n^{O(\log n)}) \subseteq \mathbf{EXP}$ using the universal Turing machine (or the efficient universal NAND-RAM program of Theorem 13.7). On the other hand, we can use similar ideas to those used to show the uncomputability of *HALT* in Section 9.3.2 to prove that $F \notin \mathbf{P}$.

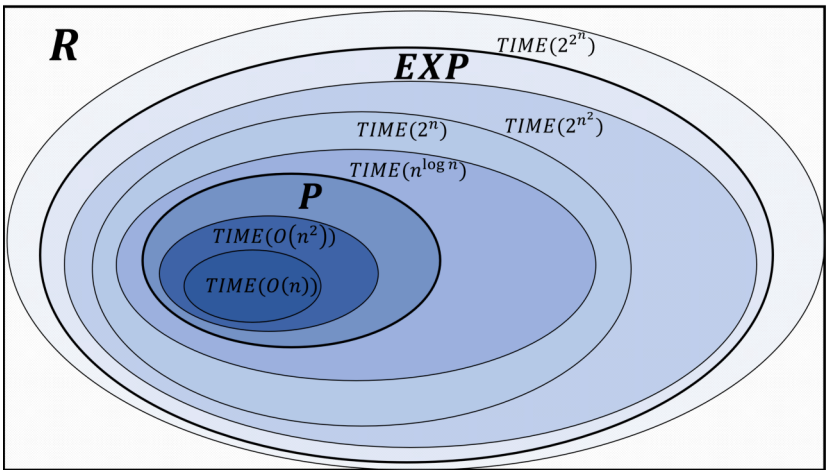


Figure 13.7: The *Time Hierarchy Theorem* (Theorem 13.9) states that all of these classes are *distinct*.

Proof Idea:

In the proof of Theorem 9.6 (the uncomputability of the Halting problem), we have shown that the function *HALT* cannot be computed in any finite time. An examination of the proof shows that it gives something stronger. Namely, the proof shows that if we fix our computational budget to be T steps, then not only can we not distinguish between programs that halt and those that do not, but we cannot even distinguish between programs that halt within at most T' steps and those that take more than that (where T' is some number depending on T). Therefore, the proof of Theorem 13.9 follows the

ideas of the uncomputability of the halting problem, but again with a more careful accounting of the running time.

⋆

Proof of Theorem 13.9. Our proof is inspired by the proof of the uncomputability of the halting problem. Specifically, for every function T as in the theorem's statement, we define the *Bounded Halting* function $HALT_T$ as follows. The input to $HALT_T$ is a pair (P, x) such that $|P| \leq \log\log|x|$ and P encodes some NAND-RAM program. We define

$$HALT_T(P,x) = \begin{cases} 1, & P \text{ halts on } x \text{ within } \leq 100 \cdot T(|P|+|x|) \text{ steps} \\ 0, & \text{otherwise} \end{cases} .$$

(The constant 100 and the function $\log\log n$ are rather arbitrary, and are chosen for convenience in this proof.)

Theorem 13.9 is an immediate consequence of the following two claims:

Claim 1: $HALT_T \in TIME(T(n)\cdot\log n)$

and

Claim 2: $HALT_T \notin TIME(T(n))$.

Please make sure you understand why indeed the theorem follows directly from the combination of these two claims. We now turn to proving them.

Proof of claim 1: We can easily check in linear time whether an input has the form P, x where $|P| \leq \log\log|x|$. Since $T(\cdot)$ is a nice function, we can evaluate it in $O(T(n))$ time. Thus, we can compute $HALT_T(P,x)$ as follows:

1. Compute $T_0 = T(|P|+|x|)$ in $O(T_0)$ steps.
2. Use the universal NAND-RAM program of Theorem 13.7 to simulate $100\cdot T_0$ steps of P on the input x using at most $poly(|P|)T_0$ steps. (Recall that we use $poly(\ell)$ to denote a quantity that is bounded by $a\ell^b$ for some constants a, b.)
3. If P halts within these $100\cdot T_0$ steps then output 1, else output 0.

The length of the input is $n = |P|+|x|$. Since $|x| \leq n$ and $(\log\log|x|)^b = o(\log|x|)$ for every b, the running time will be $o(T(|P|+|x|)\log n)$ and hence the above algorithm demonstrates that $HALT_T \in TIME(T(n)\cdot\log n)$, completing the proof of Claim 1.

Proof of claim 2: This proof is the heart of Theorem 13.9, and is very reminiscent of the proof that *HALT* is not computable. Assume, for the sake of contradiction, that there is some NAND-RAM program

P^* that computes $HALT_T(P,x)$ within $T(|P|+|x|)$ steps. We are going to show a contradiction by creating a program Q and showing that under our assumptions, if Q runs for less than $T(n)$ steps when given (a padded version of) its own code as input then it actually runs for more than $T(n)$ steps and vice versa. (It is worth re-reading the last sentence twice or thrice to make sure you understand this logic. It is very similar to the direct proof of the uncomputability of the halting problem where we obtained a contradiction by using an assumed "halting solver" to construct a program that, given its own code as input, halts if and only if it does not halt.)

We will define Q^* to be the program that on input a string z does the following:

1. If z does not have the form $z = P1^m$ where P represents a NAND-RAM program and $|P| < 0.1 \log\log m$ then return 0. (Recall that 1^m denotes the string of m ones.)

2. Compute $b = P^*(P,z)$ (at a cost of at most $T(|P|+|z|)$ steps, under our assumptions).

3. If $b = 1$ then Q^* goes into an infinite loop, otherwise it halts.

Let ℓ be the length description of Q^* as a string, and let m be larger than $2^{2^{1000\ell}}$. We will reach a contradiction by splitting into cases according to whether or not $HALT_T(Q^*, Q^*1^m)$ equals 0 or 1.

On the one hand, if $HALT_T(Q^*, Q^*1^m) = 1$, then under our assumption that P^* computes $HALT_T$, Q^* will go into an infinite loop on input $z = Q^*1^m$, and hence in particular Q^* does *not* halt within $100T(|Q^*|+m)$ steps on the input z. But this contradicts our assumption that $HALT_T(Q^*, Q^*1^m) = 1$.

This means that it must hold that $HALT_T(Q^*, Q^*1^m) = 0$. But in this case, since we assume P^* computes $HALT_T$, Q^* does not do anything in phase 3 of its computation, and so the only computation costs come in phases 1 and 2 of the computation. It is not hard to verify that Phase 1 can be done in linear and in fact less than $5|z|$ steps. Phase 2 involves executing P^*, which under our assumption requires $T(|Q^*|+m)$ steps. In total we can perform both phases in less than $10T(|Q^*|+m)$ in steps, which by definition means that $HALT_T(Q^*, Q^*1^m) = 1$, but this is of course a contradiction. This completes the proof of Claim 2 and hence of Theorem 13.9.

■

Solved Exercise 13.3 — P vs EXP. Prove that $\mathbf{P} \subsetneq \mathbf{EXP}$.

■

Solution:

We show why this statement follows from the time hierarchy theorem, but it can be an instructive exercise to prove it directly, see Remark 13.10. We need to show that there exists $F \in \mathbf{EXP} \setminus \mathbf{P}$. Let $T(n) = n^{\log n}$ and $T'(n) = n^{\log n/2}$. Both are nice functions. Since $T(n)/T'(n) = \omega(\log n)$, by Theorem 13.9 there exists some F in $TIME(T(n))/TIME(T'(n))$. Since for sufficiently large n, $2^n > n^{\log n}$, $F \in TIME(2^n) \subseteq \mathbf{EXP}$. On the other hand, $F \notin \mathbf{P}$. Indeed, suppose otherwise that there was a constant $c > 0$ and a Turing machine computing F on n-length input in at most n^c steps for all sufficiently large n. Then since for n large enough $n^c < n^{\log n/2}$, it would have followed that $F \in TIME(n^{\log n/2})$ contradicting our choice of F.

■

The time hierarchy theorem tells us that there are functions we can compute in $O(n^2)$ time but not $O(n)$, in 2^n time, but not $2^{\sqrt{n}}$, etc.. In particular there are most definitely functions that we can compute in time 2^n but not $O(n)$. We have seen that we have no shortage of natural functions for which the best *known* algorithm requires roughly 2^n time, and that many people have invested significant effort in trying to improve that. However, unlike in the finite vs. infinite case, for all of the examples above at the moment we do not know how to rule out even an $O(n)$ time algorithm. We will however see that there is a single unproven conjecture that would imply such a result for most of these problems.

The time hierarchy theorem relies on the existence of an efficient universal NAND-RAM program, as proven in Theorem 13.7. For other models such as Turing machines we have similar time hierarchy results showing that there are functions computable in time $T(n)$ and not in time $T(n)/f(n)$ where $f(n)$ corresponds to the overhead in the corresponding universal machine.

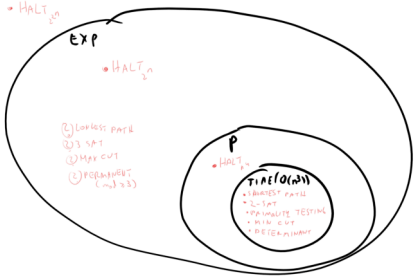


Figure 13.8: Some complexity classes and some of the functions we know (or conjecture) to be contained in them.

13.6 NON-UNIFORM COMPUTATION

We have now seen two measures of "computation cost" for functions. In Section 4.6 we defined the complexity of computing *finite* functions using circuits / straightline programs. Specifically, for a finite function $g : \{0,1\}^n \to \{0,1\}$ and number $s \in \mathbb{N}$, $g \in SIZE_n(s)$ if there is a circuit of at most s NAND gates (or equivalently an s-line NAND-CIRC program) that computes g. To relate this to the classes $TIME(T(n))$ defined in this chapter we first need to extend the class $SIZE_n(s)$ from finite functions to functions with unbounded input length.

Definition 13.11 — Non-uniform computation. Let $F : \{0,1\}^* \to \{0,1\}$ and $T : \mathbb{N} \to \mathbb{N}$ be a nice time bound. For every $n \in \mathbb{N}$, define $F_{\restriction n} : \{0,1\}^n \to \{0,1\}$ to be the *restriction* of F to inputs of size n. That is, $F_{\restriction n}$ is the function mapping $\{0,1\}^n$ to $\{0,1\}$ such that for every $x \in \{0,1\}^n$, $F_{\restriction n}(x) = F(x)$.

We say that F is *non-uniformly computable in at most $T(n)$ size*, denoted by $F \in SIZE(T)$ if there exists a sequence $(C_0, C_1, C_2, ...)$ of NAND circuits such that:

- For every $n \in \mathbb{N}$, C_n computes the function $F_{\restriction n}$
- For every sufficiently large n, C_n has at most $T(n)$ gates.

In other words, $F \in SIZE(T)$ iff for every $n \in \mathbb{N}$, it holds that $F_{\restriction n} \in SIZE_n(T(n))$. The non-uniform analog to the class **P** is the class $\mathbf{P}_{/\mathbf{poly}}$ defined as

$$\mathbf{P}_{/\mathbf{poly}} = \cup_{c \in \mathbb{N}} SIZE(n^c) \ . \tag{13.2}$$

There is a big difference between non-uniform computation and uniform complexity classes such as $TIME(T(n))$ or **P**. The condition $F \in \mathbf{P}$ means that there is a *single* Turing machine M that computes F on all inputs in polynomial time. The condition $F \in \mathbf{P}_{/\mathbf{poly}}$ only means that for every input length n there can be a *different* circuit C_n that computes F using polynomially many gates on inputs of these lengths. As we will see, $F \in \mathbf{P}_{/\mathbf{poly}}$ does not necessarily imply that $F \in \mathbf{P}$. However, the other direction is true:

Theorem 13.12 — Non-uniform computation contains uniform computation. There is some $a \in \mathbb{N}$ s.t. for every nice $T : \mathbb{N} \to \mathbb{N}$ and $F : \{0,1\}^* \to \{0,1\}$,

$$TIME(T(n)) \subseteq SIZE(T(n)^a) \ .$$

In particular, Theorem 13.12 shows that for every c, $TIME(n^c) \subseteq SIZE(n^{ca})$ and hence $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$.

Proof Idea:

The idea behind the proof is to "unroll the loop". Specifically, we will use the programming language variants of non-uniform and uniform computation: namely NAND-CIRC and NAND-TM. The main difference between the two is that NAND-TM has *loops*. However, for every fixed n, if we know that a NAND-TM program runs in at most $T(n)$ steps, then we can replace its loop by simply "copying and pasting" its code $T(n)$ times, similar to how in Python we can replace code such as

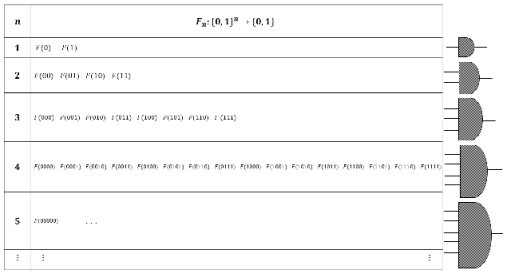


Figure 13.9: We can think of an infinite function $F : \{0,1\}^* \to \{0,1\}$ as a collection of finite functions $F_0, F_1, F_2, ...$ where $F_{\restriction n} : \{0,1\}^n \to \{0,1\}$ is the restriction of F to inputs of length n. We say F is in $\mathbf{P}_{/\mathbf{poly}}$ if for every n, the function $F_{\restriction n}$ is computable by a polynomial-size NAND-CIRC program, or equivalently, a polynomial-sized Boolean circuit.

```
for i in range(4):
    print(i)
```

with the "loop free" code

```
print(0)
print(1)
print(2)
print(3)
```

To make this idea into an actual proof we need to tackle one technical difficulty, and this is to ensure that the NAND-TM program is *oblivious* in the sense that the value of the index variable `i` in the j-th iteration of the loop will depend only on j and not on the contents of the input. We make a digression to do just that in Section 13.6.1 and then complete the proof of Theorem 13.12.

⋆

13.6.1 Oblivious NAND-TM programs

Our approach for proving Theorem 13.12 involves "unrolling the loop". For example, consider the following NAND-TM to compute the *XOR* function on inputs of arbitrary length:

```
temp_0 = NAND(X[0],X[0])
Y_nonblank[0] = NAND(X[0],temp_0)
temp_2 = NAND(X[i],Y[0])
temp_3 = NAND(X[i],temp_2)
temp_4 = NAND(Y[0],temp_2)
Y[0] = NAND(temp_3,temp_4)
MODANDJUMP(X_nonblank[i],X_nonblank[i])
```

Setting (as an example) $n = 3$, we can attempt to translate this NAND-TM program into a NAND-CIRC program for computing $XOR_3 : \{0,1\}^3 \to \{0,1\}$ by simply "copying and pasting" the loop three times (dropping the `MODANDJMP` line):

```
temp_0 = NAND(X[0],X[0])
Y_nonblank[0] = NAND(X[0],temp_0)
temp_2 = NAND(X[i],Y[0])
temp_3 = NAND(X[i],temp_2)
temp_4 = NAND(Y[0],temp_2)
Y[0] = NAND(temp_3,temp_4)
temp_0 = NAND(X[0],X[0])
Y_nonblank[0] = NAND(X[0],temp_0)
temp_2 = NAND(X[i],Y[0])
temp_3 = NAND(X[i],temp_2)
```

```
temp_4 = NAND(Y[0],temp_2)
Y[0] = NAND(temp_3,temp_4)
temp_0 = NAND(X[0],X[0])
Y_nonblank[0] = NAND(X[0],temp_0)
temp_2 = NAND(X[i],Y[0])
temp_3 = NAND(X[i],temp_2)
temp_4 = NAND(Y[0],temp_2)
Y[0] = NAND(temp_3,temp_4)
```

However, the above is still not a valid NAND-CIRC program since it contains references to the special variable `i`. To make it into a valid NAND-CIRC program, we replace references to `i` in the first iteration with 0, references in the second iteration with 1, and references in the third iteration with 2. (We also create a variable `zero` and use it for the first time any variable is instantiated, as well as remove assignments to non-output variables that are never used later on.) The resulting program is a standard "loop free and index free" NAND-CIRC program that computes XOR_3 (see also Fig. 13.10):

```
temp_0 = NAND(X[0],X[0])
one = NAND(X[0],temp_0)
zero = NAND(one,one)
temp_2 = NAND(X[0],zero)
temp_3 = NAND(X[0],temp_2)
temp_4 = NAND(zero,temp_2)
Y[0] = NAND(temp_3,temp_4)
temp_2 = NAND(X[1],Y[0])
temp_3 = NAND(X[1],temp_2)
temp_4 = NAND(Y[0],temp_2)
Y[0] = NAND(temp_3,temp_4)
temp_2 = NAND(X[2],Y[0])
temp_3 = NAND(X[2],temp_2)
temp_4 = NAND(Y[0],temp_2)
Y[0] = NAND(temp_3,temp_4)
```

Key to this transformation was the fact that in our original NAND-TM program for *XOR*, regardless of whether the input is 011, 100, or any other string, the index variable `i` is guaranteed to equal 0 in the first iteration, 1 in the second iteration, 2 in the third iteration, and so on and so forth. The particular sequence $0, 1, 2, \ldots$ is immaterial: the crucial property is that the NAND-TM program for *XOR* is *oblivious* in the sense that the value of the index `i` in the j-th iteration depends only on j and does not depend on the particular choice of the input. Luckily, it is possible to transform every NAND-TM program into a functionally equivalent oblivious program with at most quadratic

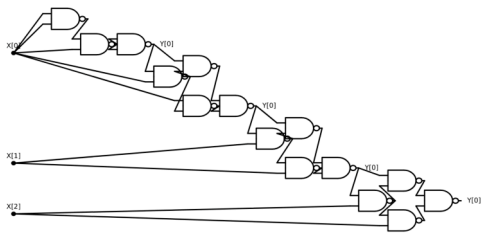


Figure 13.10: A NAND circuit for XOR_3 obtained by "unrolling the loop" of the NAND-TM program for computing *XOR* three times.

overhead. (Similarly we can transform any Turing machine into a functionally equivalent oblivious Turing machine, see Exercise 13.6.)

Theorem 13.13 — Making NAND-TM oblivious. Let $T : \mathbb{N} \to \mathbb{N}$ be a nice function and let $F \in TIME_{\mathsf{TM}}(T(n))$. Then there is a NAND-TM program P that computes F in $O(T(n)^2)$ steps and satisfying the following. For every $n \in \mathbb{N}$ there is a sequence $i_0, i_1, \ldots, i_{m-1}$ such that for every $x \in \{0,1\}^n$, if P is executed on input x then in the j-th iteration the variable `i` is equal to i_j.

In other words, Theorem 13.13 implies that if we can compute F in $T(n)$ steps, then we can compute it in $O(T(n)^2)$ steps with a program P in which the position of `i` in the j-th iteration depends only on j and the length of the input, and not on the contents of the input. Such a program can be easily translated into a NAND-CIRC program of $O(T(n)^2)$ lines by "unrolling the loop".

Proof Idea:

We can translate any NAND-TM program P' into an oblivious program P by making P "sweep" its arrays. That is, the index `i` in P will always move all the way from position 0 to position $T(n)-1$ and back again. We can then simulate the program P' with at most $T(n)$ overhead: if P' wants to move `i` left when we are in a rightward sweep then we simply wait the at most $2T(n)$ steps until the next time we are back in the same position while sweeping to the left.

⋆

Proof of Theorem 13.13. Let P' be a NAND-TM program computing F in $T(n)$ steps. We construct an oblivious NAND-TM program P for computing F as follows (see also Fig. 13.11).

1. On input x, P will compute $T = T(|x|)$ and set up arrays `Atstart` and `Atend` satisfying `Atstart[0]`$= 1$ and `Atstart[`i`]`$= 0$ for $i > 0$ and `Atend[`$T-1$`]`$= 1$ and `Atend[i]`$= 0$ for all $i \neq T-1$. We can do this because T is a nice function. Note that since this computation does not depend on x but only on its length, it is oblivious.

2. P will also have a special array `Marker` initialized to all zeroes.

3. The index variable of P will change direction of movement to the right whenever `Atstart[i]`$= 1$ and to the left whenever `Atend[i]`$= 1$.

4. The program P simulates the execution of P'. However, if the `MODANDJMP` instruction in P' attempts to move to the right when P is moving left (or vice versa) then P will set `Marker[i]` to 1 and enter into a special "waiting mode". In this mode P will wait until

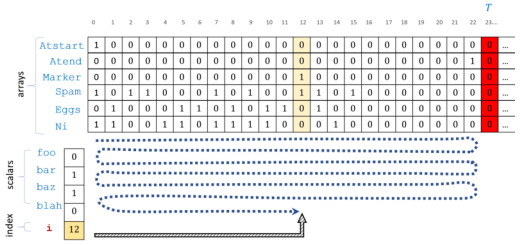


Figure 13.11: We simulate a $T(n)$-time NAND-TM program P' with an *oblivious* NAND-TM program P by adding special arrays `Atstart` and `Atend` to mark positions 0 and $T-1$ respectively. The program P will simply "sweep" its arrays from right to left and back again. If the original program P' would have moved `i` in a different direction then we wait $O(T)$ steps until we reach the same point back again, and so P runs in $O(T(n)^2)$ time.

the next time in which `Marker[i]`$= 1$ (at the next sweep) at which points P zeroes `Marker[i]` and continues with the simulation. In the worst case this will take $2T(n)$ steps (if P has to go all the way from one end to the other and back again.)

5. We also modify P to ensure it ends the computation after simulating exactly $T(n)$ steps of P', adding "dummy steps" if P' ends early.

We see that P simulates the execution of P' with an overhead of $O(T(n))$ steps of P per one step of P', hence completing the proof. ■

Theorem 13.13 implies Theorem 13.12. Indeed, if P is a k-line oblivious NAND-TM program computing F in time $T(n)$ then for every n we can obtain a NAND-CIRC program of $(k-1)\cdot T(n)$ lines by simply making $T(n)$ copies of P (dropping the final `MODANDJMP` line). In the j-th copy we replace all references of the form `Foo[i]` to `foo_`i_j where i_j is the value of `i` in the j-th iteration.

13.6.2 "Unrolling the loop": algorithmic transformation of Turing Machines to circuits

The proof of Theorem 13.12 is *algorithmic*, in the sense that the proof yields a polynomial-time algorithm that given a Turing machine M and parameters T and n, produces a circuit of $O(T^2)$ gates that agrees with M on all inputs $x \in \{0,1\}^n$ (as long as M runs for less than T steps these inputs.) We record this fact in the following theorem, since it will be useful for us later on:

Theorem 13.14 — Turing-machine to circuit compiler. There is algorithm *UNROLL* such that for every Turing machine M and numbers n, T, $\mathit{UNROLL}(M, 1^T, 1^n)$ runs for $poly(|M|, T, n)$ steps and outputs a NAND circuit C with n inputs, $O(T^2)$ gates, and one output, such that

$$C(x) = \begin{cases} y & M \text{ halts in} \leq T \text{ steps and outputs } y \\ 0 & \text{otherwise} \end{cases} .$$

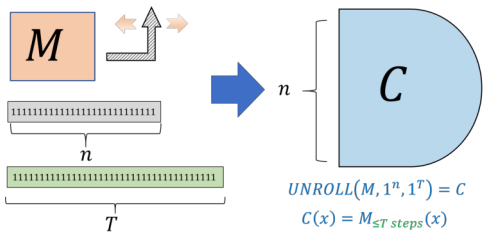


Figure 13.12: The function *UNROLL* takes as input a Turing machine M, an input length parameter n, a step budget parameter T, and outputs a circuit C of size $poly(T)$ that takes n bits of inputs and outputs $M(x)$ if M halts on x within at most T steps.

Proof. We only sketch the proof since it follows by directly translating the proof of Theorem 13.12 into an algorithm together with the simulation of Turing machines by NAND-TM programs (see also Fig. 13.13). Specifically, *UNROLL* does the following:

1. Transform the Turing machine M into an equivalent NAND-TM program P.

2. Transform the NAND-TM program P into an equivalent oblivious program P' following the proof of Theorem 13.13. The program P' takes $T' = O(T^2)$ steps to simulate T steps of P.

3. "Unroll the loop" of P' by obtaining a NAND-CIRC program of $O(T')$ lines (or equivalently a NAND circuit with $O(T^2)$ gates) corresponding to the execution of T' iterations of P'.

■

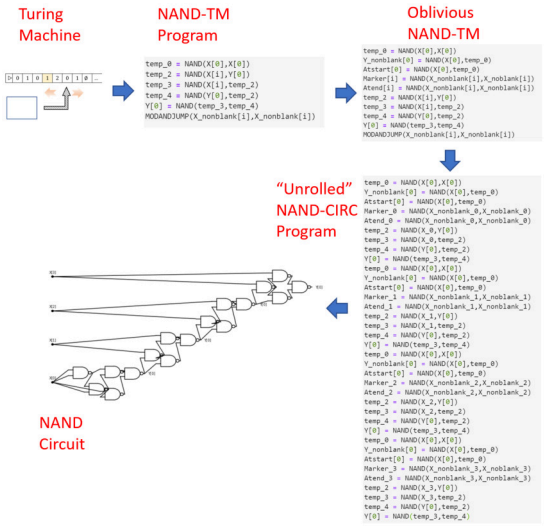


Figure 13.13: We can transform a Turing machine M, input length parameter n, and time bound T into an $O(T^2)$-sized NAND circuit that agrees with M on all inputs $x \in \{0,1\}^n$ on which M halts in at most T steps. The transformation is obtained by first using the equivalence of Turing machines and NAND-TM programs P, then turning P into an equivalent *oblivious* NAND-TM program P' via Theorem 13.13, then "unrolling" $O(T^2)$ iterations of the loop of P' to obtain an $O(T^2)$ line NAND-CIRC program that agrees with P' on length n inputs, and finally translating this program into an equivalent circuit.

Big Idea 20 By "unrolling the loop" we can transform an algorithm that takes $T(n)$ steps to compute F into a circuit that uses $poly(T(n))$ gates to compute the restriction of F to $\{0,1\}^n$.

Reviewing the transformations described in Fig. 13.13, as well as solving the following two exercises is a great way to get more comfort with non-uniform complexity and in particular with $\mathbf{P}_{/\mathbf{poly}}$ and its relation to $\mathbf{P}$.

Solved Exercise 13.4 — Alternative characterization of P. Prove that for every $F : \{0,1\}^* \to \{0,1\}$, $F \in \mathbf{P}$ if and only if there is a polynomial-time Turing machine M such that for every $n \in \mathbb{N}$, $M(1^n)$ outputs a description of an n input circuit C_n that computes the restriction $F_{\restriction n}$ of F to inputs in $\{0,1\}^n$.

■

Solution:

We start with the "if" direction. Suppose that there is a polynomial-time Turing machine M that on input 1^n outputs a circuit C_n that computes $F_{\restriction n}$. Then the following is a polynomial-time Turing machine M' to compute F. On input $x \in \{0,1\}^*$, M' will:

1. Let $n = |x|$ and compute $C_n = M(1^n)$.
2. Return the evaluation of C_n on x.

Since we can evaluate a Boolean circuit on an input in polynomial time, M' runs in polynomial time and computes $F(x)$ on every input x.

For the "only if" direction, if M' is a Turing machine that computes F in polynomial-time, then (applying the equivalence of Turing machines and NAND-TM as well as Theorem 13.13) there is also an oblivious NAND-TM program P that computes F in time $p(n)$ for some polynomial p. We can now define M to be the Turing machine that on input 1^n outputs the NAND circuit obtained by "unrolling the loop" of P for $p(n)$ iterations. The resulting NAND circuit computes $F_{\restriction n}$ and has $O(p(n))$ gates. It can also be transformed to a Boolean circuit with $O(p(n))$ AND/OR/NOT gates.

■

Solved Exercise 13.5 — $\mathbf{P}_{/\mathbf{poly}}$ characterization by advice. Let $F : \{0,1\}^* \to \{0,1\}$. Then $F \in \mathbf{P}_{/\mathbf{poly}}$ if and only if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$, a polynomial-time Turing machine M and a sequence $\{a_n\}_{n\in\mathbb{N}}$ of strings, such that for every $n \in \mathbb{N}$:

- $|a_n| \leq p(n)$
- For every $x \in \{0,1\}^n$, $M(a_n, x) = F(x)$.

■

Solution:

We only sketch the proof. For the "only if" direction, if $F \in \mathbf{P}_{/\mathbf{poly}}$ then we can use for a_n simply the description of the corresponding circuit C_n and for M the program that computes in polynomial time the evaluation of a circuit on its input.

For the "if" direction, we can use the same "unrolling the loop" technique of Theorem 13.12 to show that if P is a polynomial-time NAND-TM program, then for every $n \in \mathbb{N}$, the map $x \mapsto P(a_n, x)$ can be computed by a polynomial-size NAND-CIRC program Q_n.

■

13.6.3 Can uniform algorithms simulate non-uniform ones?

Theorem 13.12 shows that every function in $TIME(T(n))$ is in $SIZE(poly(T(n)))$. One can ask if there is an inverse relation. Suppose that F is such that $F_{\restriction n}$ has a "short" NAND-CIRC program for every n. Can we say that it must be in $TIME(T(n))$ for some "small" T? The answer is an emphatic **no**. Not only is $\mathbf{P_{/poly}}$ not contained in $\mathbf{P}$, in fact $\mathbf{P_{/poly}}$ contains functions that are *uncomputable*!

Theorem 13.15 — $\mathbf{P_{/poly}}$ contains uncomputable functions. There exists an *uncomputable* function $F : \{0,1\}^* \to \{0,1\}$ such that $F \in \mathbf{P_{/poly}}$.

Proof Idea:

Since $\mathbf{P_{/poly}}$ corresponds to non-uniform computation, a function F is in $\mathbf{P_{/poly}}$ if for every $n \in \mathbb{N}$, the restriction $F_{\restriction n}$ to inputs of length n has a small circuit/program, even if the circuits for different values of n are completely different from one another. In particular, if F has the property that for every equal-length inputs x and x', $F(x) = F(x')$ then this means that $F_{\restriction n}$ is either the constant function zero or the constant function one for every $n \in \mathbb{N}$. Since the constant function has a (very!) small circuit, such a function F will always be in $\mathbf{P_{/poly}}$ (indeed even in smaller classes). Yet by a reduction from the Halting problem, we can obtain a function with this property that is uncomputable.

⋆

Proof of Theorem 13.15. Consider the following "unary halting function" $UH : \{0,1\}^* \to \{0,1\}$ defined as follows. We let $S : \mathbb{N} \to \{0,1\}^*$ be the function that on input $n \in \mathbb{N}$, outputs the string that corresponds to the binary representation of the number n without the most significant 1 digit. Note that S is *onto*. For every $x \in \{0,1\}^*$, we define $UH(x) = HALTONZERO(S(|x|))$. That is, if n is the length of x, then $UH(x) = 1$ if and only if the string $S(n)$ encodes a NAND-TM program that halts on the input 0.

UH is uncomputable, since otherwise we could compute *HALTONZERO* by transforming the input program P into the integer n such that $P = S(n)$ and then running $UH(1^n)$ (i.e., *UH* on the string of n ones). On the other hand, for every n, $UH_n(x)$ is either equal to 0 for all inputs x or equal to 1 on all inputs x, and hence can be computed by a NAND-CIRC program of a *constant* number of lines. ■

The issue here is of course *uniformity*. For a function $F : \{0,1\}^* \to \{0,1\}$, if F is in $TIME(T(n))$ then we have a *single* algorithm that can compute $F_{\restriction n}$ for every n. On the other hand, $F_{\restriction n}$ might be in

$SIZE(T(n))$ for every n using a completely different algorithm for every input length. For this reason we typically use $\mathbf{P_{/poly}}$ not as a model of *efficient* computation but rather as a way to model *inefficient computation*. For example, in cryptography people often define an encryption scheme to be secure if breaking it for a key of length n requires more than a polynomial number of NAND lines. Since $\mathbf{P} \subseteq \mathbf{P_{/poly}}$, this in particular precludes a polynomial time algorithm for doing so, but there are technical reasons why working in a non-uniform model makes more sense in cryptography. It also allows to talk about security in non-asymptotic terms such as a scheme having "128 bits of security".

While it can sometimes be a real issue, in many natural settings the difference between uniform and non-uniform computation does not seem so important. In particular, in all the examples of problems not known to be in **P** we discussed before: longest path, 3SAT, factoring, etc., these problems are also not known to be in $\mathbf{P_{/poly}}$ either. Thus, for "natural" functions, if you pretend that $TIME(T(n))$ is roughly the same as $SIZE(T(n))$, you will be right more often than wrong.

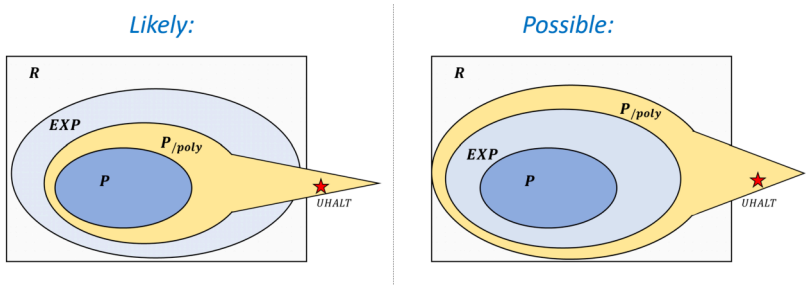


Figure 13.14: Relations between **P**, **EXP**, and $\mathbf{P_{/poly}}$. It is known that $\mathbf{P} \subseteq \mathbf{EXP}$, $\mathbf{P} \subseteq \mathbf{P_{/poly}}$ and that $\mathbf{P_{/poly}}$ contains uncomputable functions (which in particular are outside of **EXP**). It is not known whether or not $\mathbf{EXP} \subseteq \mathbf{P_{/poly}}$ though it is believed that $\mathbf{EXP} \not\subseteq \mathbf{P_{/poly}}$.

13.6.4 Uniform vs. Non-uniform computation: A recap

To summarize, the two models of computation we have described so far are:

- **Uniform models:** *Turing machines*, *NAND-TM programs*, *RAM machines*, *NAND-RAM programs*, *C*/*JavaScript*/*Python*, etc. These models include loops and unbounded memory hence a single program can compute a function with unbounded input length.

- **Non-uniform models:** *Boolean Circuits* or *straightline programs* have no loops and can only compute finite functions. The time to execute them is exactly the number of lines or gates they contain.

For a function $F : \{0,1\}^* \rightarrow \{0,1\}$ and some nice time bound $T : \mathbb{N} \rightarrow \mathbb{N}$, we know that:

- If F is uniformly computable in time $T(n)$ then there is a sequence of circuits $C_1, C_2, \ldots$ where C_n has $poly(T(n))$ gates and computes $F_{\restriction n}$ (i.e., restriction of F to $\{0,1\}^n$) for every n.
- The reverse direction is not necessarily true - there are examples of functions $F : \{0,1\}^n \rightarrow \{0,1\}$ such that $F_{\restriction n}$ can be computed by even a constant size circuit but F is uncomputable.

This means that non-uniform complexity is more useful to establish *hardness* of a function than its *easiness*.

Chapter Recap

- We can define the time complexity of a function using NAND-TM programs, and similarly to the notion of computability, this appears to capture the inherent complexity of the function.
- There are many natural problems that have polynomial-time algorithms, and other natural problems that we'd love to solve, but for which the best known algorithms are exponential.
- The definition of polynomial time, and hence the class **P**, is robust to the choice of model, whether it is Turing machines, NAND-TM, NAND-RAM, modern programming languages, and many other models.
- The time hierarchy theorem shows that there are *some* problems that can be solved in exponential, but not in polynomial time. However, we do not know if that is the case for the natural examples that we described in this lecture.
- By "unrolling the loop" we can show that every function computable in time $T(n)$ can be computed by a sequence of NAND-CIRC programs (one for every input length) each of size at most $poly(T(n))$.

13.7 EXERCISES

Exercise 13.1 — Equivalence of different definitions of P and EXP.. Prove that the classes **P** and **EXP** defined in Definition 13.2 are equal to $\cup_{c\in\{1,2,3,\ldots\}} TIME(n^c)$ and $\cup_{c\in\{1,2,3,\ldots\}} TIME(2^{n^c})$ respectively. (If $S_1, S_2, S_3, \ldots$ is a collection of sets then the set $S = \cup_{c\in\{1,2,3,\ldots\}} S_c$ is the set of all elements e such that there exists some $c \in \{1,2,3,\ldots\}$ where $e \in S_c$.)

■

Exercise 13.2 — Robustness to representation. Theorem 13.5 shows that the classes **P** and **EXP** are *robust* with respect to variations in the choice of the computational model. This exercise shows that these classes

are also robust with respect to our choice of the representation of the input.

Specifically, let F be a function mapping graphs to $\{0,1\}$, and let $F', F'' : \{0,1\}^* \to \{0,1\}$ be the functions defined as follows. For every $x \in \{0,1\}^*$:

- $F'(x) = 1$ iff x represents a graph G via the adjacency matrix representation such that $F(G) = 1$.
- $F''(x) = 1$ iff x represents a graph G via the adjacency list representation such that $F(G) = 1$.

Prove that $F' \in \mathbf{P}$ iff $F'' \in \mathbf{P}$.

More generally, for every function $F : \{0,1\}^* \to \{0,1\}$, the answer to the question of whether $F \in \mathbf{P}$ (or whether $F \in \mathbf{EXP}$) is unchanged by switching representations, as long as transforming one representation to the other can be done in polynomial time (which essentially holds for all reasonable representations).

■

Exercise 13.3 — Boolean functions. For every function $F : \{0,1\}^* \to \{0,1\}^*$, define $Bool(F)$ to be the function mapping $\{0,1\}^*$ to $\{0,1\}$ such that on input a (string representation of a) triple (x,i,σ) with $x \in \{0,1\}^*$, $i \in \mathbb{N}$ and $\sigma \in \{0,1\}$,

$$Bool(F)(x,i,\sigma) = \begin{cases} F(x)_i & \sigma = 0, i < |F(x)| \\ 1 & \sigma = 1, i < |F(x)| \\ 0 & \text{otherwise} \end{cases}$$

where $F(x)_i$ is the i-th bit of the string $F(x)$.

Prove that for every $F : \{0,1\}^* \to \{0,1\}^*$, $Bool(F) \in \mathbf{P}$ if and only if there is a Turing Machine M and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$, on input x, M halts within $\leq p(|x|)$ steps and outputs $F(x)$.

■

Exercise 13.4 — Composition of polynomial time. Say that a (possibly non-Boolean) function $F : \{0,1\}^* \to \{0,1\}^*$ is *computable in polynomial time*, if there is a Turing Machine M and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$, on input x, M halts within $\leq p(|x|)$ steps and outputs $F(x)$. Prove that for every pair of functions $F, G : \{0,1\}^* \to \{0,1\}^*$ computable in polynomial time, their *composition* $F \circ G$, which is the function H s.t. $H(x) = F(G(x))$, is also computable in polynomial time.

■

Exercise 13.5 — Non-composition of exponential time. Say that a (possibly non-Boolean) function $F : \{0,1\}^* \to \{0,1\}^*$ is *computable in exponential*

time, if there is a Turing Machine M and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$, on input x, M halts within $\leq 2^{p(|x|)}$ steps and outputs $F(x)$. Prove that there is some $F, G : \{0,1\}^* \to \{0,1\}^*$ s.t. both F and G are computable in exponential time, but $F \circ G$ is not computable in exponential time.

■

Exercise 13.6 — Oblivious Turing Machines. We say that a Turing machine M is *oblivious* if there is some function $T : \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$ such that for every input x of length n, and $t \in \mathbb{N}$ it holds that:

- If M takes more than t steps to halt on the input x, then in the t-th step M's head will be in the position $T(n,t)$. (Note that this position depends only on the *length* of x and not its contents.)
- If M halts before the t-th step then $T(n,t) = -1$.

Prove that if $F \in \mathbf{P}$ then there exists an *oblivious* Turing machine M that computes F in polynomial time. See footnote for hint.[1]

■

Exercise 13.7 Let $EDGE : \{0,1\}^* \to \{0,1\}$ be the function such that on input a string representing a triple (L,i,j), where L is the adjacency list representation of an n vertex graph G, and i and j are numbers in $[n]$, $EDGE(L,i,j) = 1$ if the edge $\{i,j\}$ is present in the graph. $EDGE$ outputs 0 on all other inputs.

1. Prove that $EDGE \in \mathbf{P}$.
2. Let $PLANARMATRIX : \{0,1\}^* \to \{0,1\}$ be the function that on input an adjacency matrix A outputs 1 if and only if the graph represented by A is *planar* (that is, can be drawn on the plane without edges crossing one another). For this question, you can use without proof the fact that $PLANARMATRIX \in \mathbf{P}$. Prove that $PLANARLIST \in \mathbf{P}$ where $PLANARLIST : \{0,1\}^* \to \{0,1\}$ is the function that on input an adjacency list L outputs 1 if and only if L represents a planar graph.

■

Exercise 13.8 — Evaluate NAND circuits. Let $NANDEVAL : \{0,1\}^* \to \{0,1\}$ be the function such that for every string representing a pair (Q,x) where Q is an n-input 1-output NAND-CIRC (not NAND-TM!) program and $x \in \{0,1\}^n$, $NANDEVAL(Q,x) = Q(x)$. On all other inputs $NANDEVAL$ outputs 0.

Prove that $NANDEVAL \in \mathbf{P}$.

■

[1] *Hint:* This is the Turing machine analog of Theorem 13.13. We replace one step of the original TM M' computing F with a "sweep" of the obliviouss TM M in which it goes T steps to the right and then T steps to the left.

Exercise 13.9 — Find hard function. Let $NANDHARD : \{0,1\}^* \to \{0,1\}$ be the function such that on input a string representing a pair (f, s) where

- $f \in \{0,1\}^{2^n}$ for some $n \in \mathbb{N}$
- $s \in \mathbb{N}$

$NANDHARD(f, s) = 1$ if there is no NAND-CIRC program Q of at most s lines that computes the function $F : \{0,1\}^n \to \{0,1\}$ whose truth table is the string f. That is, $NANDHARD(f, s) = 1$ if for every NAND-CIRC program Q of at most s lines, there exists some $x \in \{0,1\}^n$ such that $Q(x) \neq f_x$ where f_x denote the x-the coordinate of f, using the binary representation to identify $\{0,1\}^n$ with the numbers $\{0, \ldots, 2^n - 1\}$.

1. Prove that $NANDHARD \in$ **EXP**.

2. (Challenge) Prove that there is an algorithm *FINDHARD* such that if n is sufficiently large, then $FINDHARD(1^n)$ runs in time $2^{2^{O(n)}}$ and outputs a string $f \in \{0,1\}^{2^n}$ that is the truth table of a function that is not contained in $SIZE(2^n/(1000n))$. (In other words, if f is the string output by $FINDHARD(1^n)$ then if we let $F : \{0,1\}^n \to \{0,1\}$ be the function such that $F(x)$ outputs the x-th coordinate of f, then $F \notin SIZE(2^n/(1000n))$.[2]

[2] **Hint:** Use Item 1, the existence of functions requiring exponentially hard NAND programs, and the fact that there are only finitely many functions mapping $\{0,1\}^n$ to $\{0,1\}$.

■

Exercise 13.10 Suppose that you are in charge of scheduling courses in computer science in University X. In University X, computer science students wake up late, and have to work on their startups in the afternoon, and take long weekends with their investors. So you only have two possible slots: you can schedule a course either Monday-Wednesday 11am-1pm or Tuesday-Thursday 11am-1pm.

Let $SCHEDULE : \{0,1\}^* \to \{0,1\}$ be the function that takes as input a list of courses L and a list of *conflicts* C (i.e., list of pairs of courses that cannot share the same time slot) and outputs 1 if and only if there is a "conflict free" scheduling of the courses in L, where no pair in C is scheduled in the same time slot.

More precisely, the list L is a list of strings $(c_0, \ldots, c_{n-1})$ and the list C is a list of pairs of the form (c_i, c_j). $SCHEDULE(L, C) = 1$ if and only if there exists a partition of $c_0, \ldots, c_{n-1}$ into two parts so that there is no pair $(c_i, c_j) \in C$ such that both c_i and c_j are in the same part.

Prove that $SCHEDULE \in$ **P**. As usual, you do not have to provide the full code to show that this is the case, and can describe operations as a high level, as well as appeal to any data structures or other results mentioned in the book or in lecture. Note that to show that a function F is in **P** you need to both **(1)** present an algorithm A that computes

F in polynomial time, (**2**) *prove* that A does indeed run in polynomial time, and does indeed compute the correct answer.

Try to think whether or not your algorithm extends to the case where there are *three* possible time slots.

■

13.8 BIBLIOGRAPHICAL NOTES

Because we are interested in the *maximum* number of steps for inputs of a given length, running-time as we defined it is often known as *worst case complexity*. The *minimum* number of steps (or "best case" complexity) to compute a function on length n inputs is typically not a meaningful quantity since essentially every natural problem will have some trivially easy instances. However, the *average case complexity* (i.e., complexity on a "typical" or "random" input) is an interesting concept which we'll return to when we discuss *cryptography*. That said, worst-case complexity is the most standard and basic of the complexity measures, and will be our focus in most of this book.

Some lower bounds for single-tape Turing machines are given in [Maa85].

For defining efficiency in the λ calculus, one needs to be careful about the order of application of the reduction steps, which can matter for computational efficiency, see for example this paper.

The notation $\mathbf{P}_{/\mathbf{poly}}$ is used for historical reasons. It was introduced by Karp and Lipton, who considered this class as corresponding to functions that can be computed by polynomial-time Turing machines that are given for any input length n an *advice string* of length polynomial in n.

Learning Objectives:

- Introduce the notion of *polynomial-time reductions* as a way to relate the complexity of problems to one another.
- See several examples of such reductions.
- 3SAT as a basic starting point for reductions.

14
Polynomial-time reductions

Consider some of the problems we have encountered in Chapter 12:

1. The *3SAT* problem: deciding whether a given 3CNF formula has a satisfying assignment.
2. Finding the *longest path* in a graph.
3. Finding the *maximum cut* in a graph.
4. Solving *quadratic equations* over n variables $x_0, \ldots, x_{n-1} \in \mathbb{R}$.

All of these problems have the following properties:

- These are important problems, and people have spent significant effort on trying to find better algorithms for them.
- Each one of these is a *search* problem, whereby we search for a solution that is "good" in some easy to define sense (e.g., a long path, a satisfying assignment, etc.).
- Each of these problems has a trivial exponential time algorithm that involve enumerating all possible solutions.
- At the moment, for all these problems the best known algorithm is not much faster than the trivial one in the worst case.

In this chapter and in Chapter 15 we will see that, despite their apparent differences, we can relate the computational complexity of these and many other problems. In fact, it turns out that the problems above are *computationally equivalent*, in the sense that solving one of them immediately implies solving the others. This phenomenon, known as **NP** *completeness*, is one of the surprising discoveries of theoretical computer science, and we will see that it has far-reaching ramifications.

This chapter: A non-mathy overview

This chapter introduces the concept of a *polynomial time reduction* which is a central object in computational complexity and this book in particular. A polynomial-time reduction is a way to *reduce* the task of solving one problem to another. The way we use reductions in complexity is to argue that if the first problem is hard to solve efficiently, then the second must also be hard. We see several examples for reductions in this chapter, and reductions will be the basis for the theory of **NP** *completeness* that we will develop in Chapter 15.

All the code for the reductions described in this chapter is available on the following Jupyter notebook.

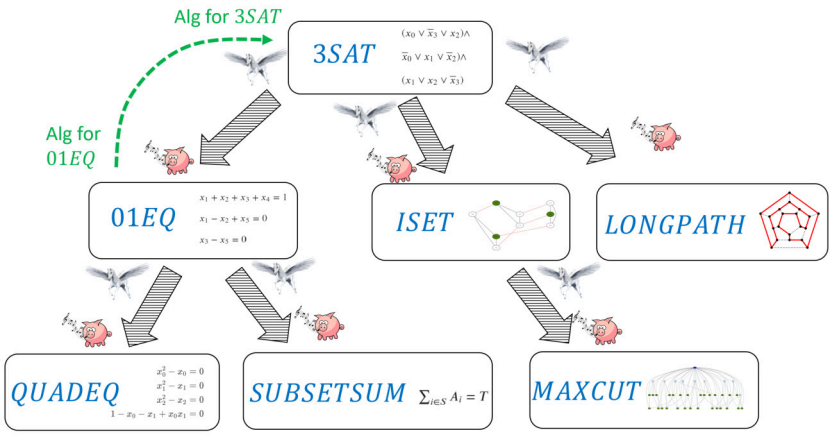


Figure 14.1: In this chapter we show that if the 3*SAT* problem cannot be solved in polynomial time, then neither can the *QUADEQ*, *LONGESTPATH*, *ISET* and *MAXCUT* problems. We do this by using the *reduction paradigm* showing for example "if pigs could whistle" (i.e., if we had an efficient algorithm for *QUADEQ*) then "horses could fly" (i.e., we would have an efficient algorithm for 3*SAT*.)

In this chapter we will see that for each one of the problems of finding a longest path in a graph, solving quadratic equations, and finding the maximum cut, if there exists a polynomial-time algorithm for this problem then there exists a polynomial-time algorithm for the 3SAT problem as well. In other words, we will *reduce* the task of solving 3SAT to each one of the above tasks. Another way to interpret these results is that if there *does not exist* a polynomial-time algorithm for 3SAT then there does not exist a polynomial-time algorithm for these other problems as well. In Chapter 15 we will see evidence (though no proof!) that all of the above problems do not have polynomial-time algorithms and hence are *inherently intractable*.

14.1 FORMAL DEFINITIONS OF PROBLEMS

For reasons of technical convenience rather than anything substantial, we concern ourselves with *decision problems* (i.e., Yes/No questions) or

in other words *Boolean* (i.e., one-bit output) functions. We model the problems above as functions mapping $\{0,1\}^*$ to $\{0,1\}$ in the following way:

3SAT. The *3SAT problem* can be phrased as the function $3SAT : \{0,1\}^* \to \{0,1\}$ that takes as input a 3CNF formula φ (i.e., a formula of the form $C_0 \wedge \cdots \wedge C_{m-1}$ where each C_i is the OR of three variables or their negation) and maps φ to 1 if there exists some assignment to the variables of φ that causes it to evalute to *true*, and to 0 otherwise. For example

$$3SAT("(x_0 \vee \overline{x}_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x}_0 \vee \overline{x}_2 \vee x_3)") = 1$$

since the assignment $x = 1101$ satisfies the input formula. In the above we assume some representation of formulas as strings, and define the function to output 0 if its input is not a valid representation; we use the same convention for all the other functions below.

Quadratic equations. The *quadratic equations problem* corresponds to the function $QUADEQ : \{0,1\}^* \to \{0,1\}$ that maps a set of quadratic equations E to 1 if there is an assignment x that satisfies all equations, and to 0 otherwise.

Longest path. The *longest path problem* corresponds to the function $LONGPATH : \{0,1\}^* \to \{0,1\}$ that maps a graph G and a number k to 1 if there is a simple path in G of length at least k, and maps (G,k) to 0 otherwise. The longest path problem is a generalization of the well-known Hamiltonian Path Problem of determining whether a path of length n exists in a given n vertex graph.

Maximum cut. The *maximum cut problem* corresponds to the function $MAXCUT : \{0,1\}^* \to \{0,1\}$ that maps a graph G and a number k to 1 if there is a cut in G that cuts at least k edges, and maps (G,k) to 0 otherwise.

All of the problems above are in **EXP** but it is not known whether or not they are in **P**. However, we will see in this chapter that if either *QUADEQ* , *LONGPATH* or *MAXCUT* are in **P**, then so is *3SAT*.

14.2 POLYNOMIAL-TIME REDUCTIONS

Suppose that $F, G : \{0,1\}^* \to \{0,1\}$ are two Boolean functions. A *polynomial-time reduction* (or sometimes just *"reduction"* for short) from F to G is a way to show that F is "no harder" than G, in the sense that a polynomial-time algorithm for G implies a polynomial-time algorithm for F.

Definition 14.1 — Polynomial-time reductions. Let $F, G : \{0,1\}^* \to \{0,1\}$. We say that F *reduces to* G, denoted by $F \leq_p G$ if there is a polynomial-time computable $R : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$,

$$F(x) = G(R(x)) \ . \tag{14.1}$$

We say that F and G have *equivalent complexity* if $F \leq_p G$ and $G \leq_p F$.

The following exercise justifies our intuition that $F \leq_p G$ signifies that "F is no harder than G".

Solved Exercise 14.1 — Reductions and P. Prove that if $F \leq_p G$ and $G \in \mathbf{P}$ then $F \in \mathbf{P}$.

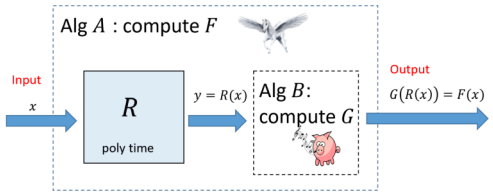


Figure 14.2: If $F \leq_p G$ then we can transform a polynomial-time algorithm B that computes G into a polynomial-time algorithm A that computes F. To compute $F(x)$ we can run the reduction R guaranteed by the fact that $F \leq_p G$ to obtain $y = R(x)$ and then run our algorithm B for G to compute $G(y)$.

As usual, solving this exercise on your own is an excellent way to make sure you understand Definition 14.1.

Solution:

Suppose there is an algorithm B that computes G in time $p(n)$ where n is its input size. Then, (14.1) directly gives an algorithm A to compute F (see Fig. 14.2). Indeed, on input $x \in \{0,1\}^*$, Algorithm A will run the polynomial-time reduction R to obtain $y = R(x)$ and then return $B(y)$. By (14.1), $G(R(x)) = F(x)$ and hence Algorithm A will indeed compute F.

We now show that A runs in polynomial time. By assumption, R can be computed in time $q(n)$ for some polynomial q. In particular, this means that $|y| \leq q(|x|)$ (as just writing down y takes $|y|$ steps). Computing $B(y)$ will take at most $p(|y|) \leq p(q(|x|))$ steps. Thus the total running time of A on inputs of length n is at most the time to compute y, which is bounded by $q(n)$, and the time to compute $B(y)$, which is bounded by $p(q(n))$, and since the composition of two polynomials is a polynomial, A runs in polynomial time.

Big Idea 21 A *reduction* $F \leq_p G$ shows that F is "no harder than G" or equivalently that G is "no easier than F".

14.2.1 Whistling pigs and flying horses

A reduction from F to G can be used for two purposes:

- If we already know an algorithm for G and $F \leq_p G$ then we can use the reduction to obtain an algorithm for F. This is a widely used tool in algorithm design. For example in Section 12.1.4 we saw how the *Min-Cut Max-Flow* theorem allows to reduce the task of computing a minimum cut in a graph to the task of computing a maximum flow in it.

- If we have proven (or have evidence) that there exists *no polynomial-time algorithm* for F and $F \leq_p G$ then the existence of this reduction allows us to conclude that there exists no polynomial-time algorithm for G. This is the "if pigs could whistle then horses could fly" interpretation we've seen in Section 9.4. We show that if there was an hypothetical efficient algorithm for G (a "whistling pig") then since $F \leq_p G$ then there would be an efficient algorithm for F (a "flying horse"). In this book we often use reductions for this second purpose, although the lines between the two is sometimes blurry (see the bibliographical notes in Section 14.10).

The most crucial difference between the notion in Definition 14.1 and the reductions we saw in the context of *uncomputability* (e.g., in Section 9.4) is that for relating time complexity of problems, we need the reduction to be computable in *polynomial time*, as opposed to merely computable. Definition 14.1 also restricts reductions to have a very specific format. That is, to show that $F \leq_p G$, rather than allowing a general algorithm for F that uses a "magic box" that computes G, we only allow an algorithm that computes $F(x)$ by outputting $G(R(x))$. This restricted form is convenient for us, but people have defined and used more general reductions as well (see Section 14.10).

In this chapter we use reductions to relate the computational complexity of the problems mentioned above: 3SAT, Quadratic Equations, Maximum Cut, and Longest Path, as well as a few others. We will reduce 3SAT to the latter problems, demonstrating that solving any one of them efficiently will result in an efficient algorithm for 3SAT. In Chapter 15 we show the other direction: reducing each one of these problems to 3SAT in one fell swoop.

Transitivity of reductions. Since we think of $F \leq_p G$ as saying that (as far as polynomial-time computation is concerned) F is "easier or equal in difficulty to" G, we would expect that if $F \leq_p G$ and $G \leq_p H$, then it would hold that $F \leq_p H$. Indeed this is the case:

Solved Exercise 14.2 — Transitivity of polynomial-time reductions. For every $F, G, H : \{0,1\}^* \to \{0,1\}$, if $F \leq_p G$ and $G \leq_p H$ then $F \leq_p H$.

■

Solution:

If $F \leq_p G$ and $G \leq_p H$ then there exist polynomial-time computable functions R_1 and R_2 mapping $\{0,1\}^*$ to $\{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $F(x) = G(R_1(x))$ and for every $y \in \{0,1\}^*$, $G(y) = H(R_2(y))$. Combining these two equalities, we see that for every $x \in \{0,1\}^*$, $F(x) = H(R_2(R_1(x)))$ and so to show that $F \leq_p H$, it is sufficient to show that the map $x \mapsto R_2(R_1(x))$ is computable in polynomial time. But if there are some constants c, d such that $R_1(x)$ is computable in time $|x|^c$ and $R_2(y)$ is computable in time $|y|^d$ then $R_2(R_1(x))$ is computable in time $(|x|^c)^d = |x|^{cd}$ which is polynomial.

■

14.3 REDUCING 3SAT TO ZERO ONE AND QUADRATIC EQUATIONS

We now show our first example of a reduction. The *Zero-One Linear Equations problem* corresponds to the function $01EQ : \{0,1\}^* \to \{0,1\}$ whose input is a collection E of linear equations in variables $x_0, \ldots, x_{n-1}$, and the output is 1 iff there is an assignment $x \in \{0,1\}^n$ of $0/1$ values to the variables that satisfies all the equations. For example, if the input E is a string encoding the set of equations

$$\begin{aligned} x_0 + x_1 + x_2 &= 2 \\ x_0 + x_2 &= 1 \\ x_1 + x_2 &= 2 \end{aligned}$$

then $01EQ(E) = 1$ since the assignment $x = 011$ satisfies all three equations. We specifically restrict attention to linear equations in variables $x_0, \ldots, x_{n-1}$ in which every equation has the form $\sum_{i \in S} x_i = b$ where $S \subseteq [n]$ and $b \in \mathbb{N}$.[1]

If we asked the question of whether there is a solution $x \in \mathbb{R}^n$ of *real numbers* to E, then this can be solved using the famous *Gaussian elimination* algorithm in polynomial time. However, there is no known efficient algorithm to solve $01EQ$. Indeed, such an algorithm would imply an algorithm for $3SAT$ as shown by the following theorem:

[1] If you are familiar with matrix notation you may note that such equations can be written as $Ax = \mathbf{b}$ where A is an $m \times n$ matrix with entries in $0/1$ and $\mathbf{b} \in \mathbb{N}^m$.

Theorem 14.2 — Hardness of $01EQ$. $3SAT \leq_p 01EQ$

Proof Idea:

A constraint $x_2 \vee \overline{x}_5 \vee x_7$ can be written as $x_2 + (1 - x_5) + x_7 \geq 1$. This is a linear *inequality* but since the sum on the left-hand side is at most three, we can also turn it into an *equality* by adding two new variables y, z and writing it as $x_2 + (1 - x_5) + x_7 + y + z = 3$. (We will use fresh variables y, z for every constraint.) Finally, for every variable x_i we can add a variable x'_i corresponding to its negation by

adding the equation $x_i + x'_i = 1$, hence mapping the original constraint $x_2 \vee \overline{x}_5 \vee x_7$ to $x_2 + x'_5 + x_7 + y + z = 3$. The main **takeaway technique** from this reduction is the idea of adding *auxiliary variables* to replace an equation such as $x_1 + x_2 + x_3 \geq 1$ that is not quite in the form we want with the equivalent (for $0/1$ valued variables) equation $x_1 + x_2 + x_3 + u + v = 3$ which is in the form we want.

⋆

```python
def SAT2ZOE(ϕ):
    # Reduce 3SAT to 0/1 equations
    n = numvars(ϕ)
    E = ""
    for i in range(n): # add vars for negations
        E+= f"x{subscript(i)} + x{subscript(n+i)} = 1\n"
    m = 2*n
    for literals in getclauses(ϕ):
        # map each clause to equation
        def var(lit): # map literal to variable
            return f"x{subscript(n+int(lit[2:]))}" if lit[0]== "¬" else f"x{subscript(lit[1:])}"
        E+= " + ".join([var(lit) for lit in literals])
        E+= f" + x{subscript(m)} + x{subscript(m+1)} = 3\n"
        m += 2
    return Equation(E)
```

```
SAT2ZOE("(x0 ∨ ¬x3 ∨ x2 ) ∧ (x0 ∨ x1 ∨ ¬x2 ) ∧ (x1 ∨ x2 ∨ ¬x3 )")
```

$$
\begin{aligned}
x_0 + x_4 &= 1 \\
x_1 + x_5 &= 1 \\
x_2 + x_6 &= 1 \\
x_3 + x_7 &= 1 \\
x_0 + x_7 + x_2 + x_8 + x_9 &= 3 \\
x_0 + x_1 + x_6 + x_{10} + x_{11} &= 3 \\
x_1 + x_2 + x_7 + x_{12} + x_{13} &= 3
\end{aligned}
$$

Figure 14.3: Left: Python code implementing the reduction of $3SAT$ to $01EQ$. Right: Example output of the reduction. Code is in our repository.

Proof of Theorem 14.2. To prove the theorem we need to:

1. Describe an algorithm R for mapping an input φ for $3SAT$ into an input E for $01EQ$.

2. Prove that the algorithm runs in polynomial time.

3. Prove that $01EQ(R(\varphi)) = 3SAT(\varphi)$ for every 3CNF formula φ.

We now proceed to do just that. Since this is our first reduction, we will spell out this proof in detail. However it straightforwardly follows the proof idea.

Algorithm 14.3 — $3SAT$ to $01EQ$ reduction.

Input: 3CNF formula φ with n variables $x_0, \ldots, x_{n-1}$ and m clauses.

Output: Set E of linear equations over $0/1$ such that $3SAT(\varphi) = 1$ iff $01EQ(E) = 1$.

Let E's variables be $x_0, \ldots, x_{n-1}, x'_0, \ldots, x'_{n-1}, y_0, \ldots, y_{m-1}, z_0, \ldots, z_{m-1}$.
for $i \in [n]$ **do**
    add to E the equation $x_i + x'_i = 1$
end for
for $j \in [m]$ **do**
    Let j-th clause be $w_0 \vee w_1 \vee w_2$ where w_0, w_1, w_2 are literals.
    for $a \in [3]$ **do**
        if w_a is variable x_i **then**
            set $t_a \leftarrow x_i$
        end if
        if w_a is negation $\neg x_i$ **then**
            set $t_a \leftarrow x'_i$
        end if
    end for
    Add to E the equation $t_0 + t_1 + t_2 + y_j + z_j = 3$.
end for
return E

The reduction is described in Algorithm 14.3, see also Fig. 14.3. If the input formula has n variables and m clauses, Algorithm 14.3 creates a set E of $n + m$ equations over $2n + 2m$ variables. Algorithm 14.3 makes an initial loop of n steps (each taking constant time) and then another loop of m steps (each taking constant time) to create the equations, and hence it runs in polynomial time.

Let R be the function computed by Algorithm 14.3. The heart of the proof is to show that for every 3CNF φ, $01EQ(R(\varphi)) = 3SAT(\varphi)$. We split the proof into two parts. The first part, traditionally known as the **completeness** property, is to show that if $3SAT(\varphi) = 1$ then $01EQ(R(\varphi)) = 1$. The second part, traditionally known as the **soundness** property, is to show that if $01EQ(R(\varphi)) = 1$ then $3SAT(\varphi) = 1$. (The names "completeness" and "soundness" derive viewing a solution to $R(\varphi)$ as a "proof" that φ is satisfiable, in which case these conditions corresponds to completeness and soundness as defined in Section 11.1.1. However, if you find the names confusing you can simply think of completeness as the "1-instance maps to 1-instance"

property and soundness as the "0-instance maps to 0-instance" property.)

We complete the proof by showing both parts:

- **Completeness:** Suppose that $3SAT(\varphi) = 1$, which means that there is an assignment $x \in \{0,1\}^n$ that satisfies φ. If we use the assignment $x_0, \ldots, x_{n-1}$ and $1 - x_0, \ldots, 1 - x_{n-1}$ for the first $2n$ variables of $E = R(\varphi)$ then we will satisfy all equations of the form $x_i + x'_i = 1$. Moreover, for every $j \in [m]$, if $t_0 + t_1 + t_2 + y_j + z_j = 3(*)$ is the equation arising from the jth clause of φ (with t_0, t_1, t_2 being variables of the form x_i or x'_i depending on the literals of the clause) then our assignment to the first $2n$ variables ensures that $t_0 + t_1 + t_2 \geq 1$ (since x satisfied φ) and hence we can assign values to y_j and z_j that will ensure that the equation $(*)$ is satisfied. Hence in this case $E = R(\varphi)$ is satisfied, meaning that $01EQ(R(\varphi)) = 1$.

- **Soundness:** Suppose that $01EQ(R(\varphi)) = 1$, which means that the set of equations $E = R(\varphi)$ has a satisfying assignment $x_0, \ldots, x_{n-1}$, $x'_0, \ldots, x'_{n-1}$, $y_0, \ldots, y_{m-1}$, $z_0, \ldots, z_{m-1}$. Then, since the equations contain the condition $x_i + x'_i = 1$, for every $i \in [n]$, x'_i is the negation of x_i, and morover, for every $j \in [m]$, if C has the form $w_0 \vee w_1 \vee w_2$ and is the j-th clause of φ, then the corresponding assignment x will ensure that $w_0 + w_1 + w_2 \geq 1$, implying that C is satisfied. Hence in this case $3SAT(\varphi) = 1$.

■

14.3.1 Quadratic equations

Now that we reduced $3SAT$ to $01EQ$, we can use this to reduce $3SAT$ to the *quadratic equations* problem. This is the function $QUADEQ$ in which the input is a list of n-variate polynomials $p_0, \ldots, p_{m-1} : \mathbb{R}^n \to \mathbb{R}$ that are all of degree at most two (i.e., they are *quadratic*) and with integer coefficients. (The latter condition is for convenience and can be achieved by scaling.) We define $QUADEQ(p_0, \ldots, p_{m-1})$ to equal 1 if and only if there is a solution $x \in \mathbb{R}^n$ to the equations $p_0(x) = 0$, $p_1(x) = 0$, ..., $p_{m-1}(x) = 0$.

For example, the following is a set of quadratic equations over the variables x_0, x_1, x_2:

$$\begin{aligned} x_0^2 - x_0 &= 0 \\ x_1^2 - x_1 &= 0 \\ x_2^2 - x_2 &= 0 \\ 1 - x_0 - x_1 + x_0 x_1 &= 0 \end{aligned}$$

You can verify that $x \in \mathbb{R}^3$ satisfies this set of equations if and only if $x \in \{0,1\}^3$ and $x_0 \vee x_1 = 1$.

Theorem 14.4 — Hardness of quadratic equations.

$$3SAT \leq_p QUADEQ$$

Proof Idea:

Using the transitivity of reductions (Solved Exercise 14.2), it is enough to show that $01EQ \leq_p QUADEQ$, but this follows since we can phrase the equation $x_i \in \{0,1\}$ as the quadratic constraint $x_i^2 - x_i = 0$. The **takeaway technique** of this reduction is that we can use *non-linearity* to force continuous variables (e.g., variables taking values in $\mathbb{R}$) to be discrete (e.g., take values in $\{0,1\}$).

⋆

Proof of Theorem 14.4. By Theorem 14.2 and Solved Exercise 14.2, it is sufficient to prove that $01EQ \leq_p QUADEQ$. Let E be an instance of $01EQ$ with variables $x_0, \ldots, x_{n-1}$. We map E to the set of quadratic equations E' that is obtained by taking the linear equations in E and adding to them the n quadratic equations $x_i^2 - x_i = 0$ for all $i \in [n]$. (See Algorithm 14.5.) The map $E \mapsto E'$ can be computed in polynomial time. We claim that $01EQ(E) = 1$ if and only if $QUADEQ(E') = 1$. Indeed, the only difference between the two instances is that:

- In the $01EQ$ instance E, the equations are over variables $x_0, \ldots, x_{n-1}$ in $\{0,1\}$.
- In the $QUADEQ$ instance E', the equations are over variables $x_0, \ldots, x_{n-1} \in \mathbb{R}$ but we have the extra constraints $x_i^2 - x_i = 0$ for all $i \in [n]$.

Since for every $a \in \mathbb{R}$, $a^2 - a = 0$ if and only if $a \in \{0,1\}$, the two sets of equations are equivalent and $01EQ(E) = QUADEQ(E')$ which is what we wanted to prove.

■

Algorithm 14.5 — $01EQ$ to $QUADEQ$ reduction.

Input: Set E of linear equations over n variables $x_0, \ldots, x_{n-1}$.
Output: Set E' of quadratic equations over m variables $w_0, \ldots, w_{m-1}$ such that there is an $0/1$ assignment $x \in \{0,1\}^n$
satisfying the equations of E iff there is an assignment $w \in \mathbb{R}^m$ satisfying the equations of E'.
That is, $01EQ(E) = QUADEQ(E')$.
Let $m \leftarrow n$.
Variables of E' are set to be same variable $x_0, \ldots, x_{n-1}$ as E.
for every equation $e \in E$ **do**
    Add e to E'
end for
for $i \in [n]$ **do**
    Add to E' the equation $x_i^2 - x_i = 0$.
end for
return E'

14.4 THE SUBSET SUM PROBLEM

As another consequence of the reduction of $3SAT$ to $01EQ$, we can also show that $3SAT$ (through $01EQ$) reduces to the *subset sum* problem (also known as the *knapsack* problem). In the *subset sum* problem, we are given a list of integers $x_0, \ldots, x_{n-1} \in \mathbb{Z}$ and an integer $T \in \mathbb{Z}$. We need to determine whether or not there exists some set of the integers that sums up to T. That is, for $x_0, \ldots, x_{n-1}, T \in \mathbb{Z}$, $SSUM(x_0, \ldots, x_{n-1}, T) = 1$ if and only if there exists $S \subseteq [n]$ such that $\sum_{i \in S} x_i = T$. Note that the input length for the subset sum problem is the length of string needed to encode all the numbers, which will be approximately $\lceil \log T \rceil + \sum_{i=0}^{n} \lceil \log x_i \rceil$, since encoding an integer x using the binary representation requires $\lceil \log x \rceil$ bits.

Theorem 14.6 — Hardness of subset sum.

$$3SAT \leq_p SSUM$$

Proof Idea:

We reduce from $01EQ$. The intuition is the following. Consider an instance E of $01EQ$ with n variables $x_0, \ldots, x_{n-1}$ and m equations $e_0, \ldots, e_{m-1}$. Recall that each equation e_ℓ in E has the form $x_i + x_j + x_k = b$ (potentially with more or less than three variables summed up on the left-hand side of the equation). For every variable x_i, we can define a vector $v^i \in \{0,1\}^m$ where $v^i_t = 1$ if the variable

x_i appears in the equation e_t and $v^i_t = 0$ otherwise. Then there is a solution to the set of equations if and only if there is some set $S \subseteq [n]$ (corresponding to the i's such that $x_i = 1$) such that $\sum_{i \in S} v^i = \vec{b}$ where $\vec{b} \in \mathbb{Z}^m$ is the vector of right hand sides of the equations (i.e., $\vec{b}_t$ is the value b_t on the righthand side of the t-th equation). Now if we could interpret the vectors $v^0, \ldots, v^{n-1}$ and $\vec{b}$ as *numbers* then we could think of this as a subset sum instance. The key insight is that we can in fact think of vectors as numbers by thinking of the j-th coordinate of the vector v as the j-th digit. Since the vectors are in $\{0,1\}^m$, the natural choice is to use the binary basis, but this turns out to cause issues with "carries" when we add them up. Hence we use a larger basis B, see proof below.

⋆

Proof of Theorem 14.6. For a given set of 01*EQ* on n variables, we note that the right hand side can never be larger than n (since the sum of at most n variables in $\{0,1\}$ is at most n). More concretely, if the instance has such an equation then we can know for sure that the answer is 0 (and in the context of a reduction map it into some trivial instance of subset sum that doesn't have a solution such as $x_0 = x_1 = 1$ and $T = 3$).

Our reduction is described in Algorithm 14.7. On input an instance $E = \{e_t\}_{t=1}^m$ of 01*EQ* over n variables $x_0, \ldots, x_{n-1}$, we output an *SSUM* instance $y_0, \ldots, y_{n-1}, T$ computed as follows:

- $y_i = \sum_{t=0}^{m-1} B^t v^t_i$ where v^t_i equals 1 if the variable x_i appears in the equation e_t and equals 0 otherwise. The number B is set to be $2n$ (any numb er larger than n would work.)
- $T = \sum_{t=0}^{m-1} B^t b_t$ where b_t is the integer on the right-hand side of the equation e_t.

In other words, $y_0, \ldots, y_{n-1}$ and T are the integers such that, written in the B-ary basis, the t-th digit of y_i is 1 iff x_i appears in x_t, and the t-th digit of T is the right-hand side of e_t.

The following claim will imply the correctness of the reduction:

Claim: For every $x \in \{0,1\}^n$, if $S = \{i | x_i = 1\}$ then x satisfies the equations of E if and only if $\sum_{i \in S} y_i = T$.

Proof: Key to the proof is the following simple property of grade-school addition: when adding at most n numbers in the B-ary basis, if all the numbers have all their digits either 0 or 1, and $B > n$, then for every t, the t-th digit of the sum is the sum of the t-th digits of the numbers. This is a simple consequence of the fact that there is no "carry" in the addition. Since in our case the numbers $y_0, \ldots, y_n$ satisfy this property in the B-ary basis, and $B > n$, we get that for every

$S \subseteq [n]$ and every digit t, the t-th digit of the sum $\sum_{i \in S} y_i$ is simply the sum of the t-th digit, which would correspond to the sum over x_i for all x_i's that participate in the t-th equation. This sum would equal the t-th digit of T if and only if that equation is satisfied.

The claim shows that $01EQ(E) = SSUM(y_0, \ldots, y_{n-1}, T)$ which is what we needed to prove.

■

Algorithm 14.7 — $01EQ$ to $SSUM$ reduction.

Input: Set $E = \{e_t\}_{t \in [m]}$ of m linear equations over n variables $x_0, \ldots, x_{n-1}$.

Output: Numbers $y_0, \ldots, y_{n-1}, T \in \mathbb{Z}$ such that there is an $0/1$ assignment $x \in \{0,1\}^n$ satisfying the equations of E iff there is $S \subseteq [n]$ such that $\sum_{i \in S} y_i = T$.

1: **for** every equation $e_t \in E$ **do**
2: Let $A \subseteq [n]$ and $b \in \mathbb{Z}$ be such that e_t has the form $\sum_{i \in A} x_i = b$
3: Let $v_i^t \leftarrow 1$ if $i \in A$ and $v_i^t \leftarrow 0$ otherwise.
4: Let $b_t \leftarrow b$.
5: **end for**
6: Set $B \leftarrow 2n$
7: **for** $i \in [n]$ **do**
8: Let $y_i \leftarrow \sum_{t=1}^{m} B^t v_i^t$.
9: **end for**
10: Let $T \leftarrow \sum_{t=1}^{T} B^t b_t$
11: **return** $y_0, \ldots, y_{n-1}, T$

14.5 THE INDEPENDENT SET PROBLEM

For a graph $G = (V, E)$, an independent set (also known as a *stable set*) is a subset $S \subseteq V$ such that there are no edges with both endpoints in S (in other words, $E(S, S) = \varnothing$). Every "singleton" (set consisting of a single vertex) is trivially an independent set, but finding larger independent sets can be challenging. The *maximum independent set* problem (henceforth simply "independent set") is the task of finding the largest independent set in the graph. The independent set problem is naturally related to *scheduling problems*: if we put an edge between two conflicting tasks, then an independent set corresponds to a set of tasks that can all be scheduled together without conflicts. The independent set problem has been studied in a variety of settings, including for example in the case of algorithms for finding structure in protein-protein interaction graphs.

As mentioned in Section 14.1, we think of the independent set problem as the function $ISET : \{0,1\}^* \to \{0,1\}$ that on input a graph G

and a number k outputs 1 if and only if the graph G contains an independent set of size at least k. We now reduce 3SAT to Independent set.

Theorem 14.8 — Hardness of Independent Set. $3SAT \leq_p ISET$.

Proof Idea:

The idea is that finding a satisfying assignment to a 3SAT formula corresponds to satisfying many local constraints without creating any conflicts. One can think of "$x_{17} = 0$" and "$x_{17} = 1$" as two conflicting events, and of the constraints $x_{17} \vee \overline{x}_5 \vee x_9$ as creating a conflict between the events "$x_{17} = 0$", "$x_5 = 1$" and "$x_9 = 0$", saying that these three cannot simultaneously co-occur. Using these ideas, we can we can think of solving a 3SAT problem as trying to schedule non-conflicting events, though the devil is, as usual, in the details. The **takeaway technique** here is to map each clause of the original formula into a *gadget* which is a small subgraph (or more generally "subinstance") satisfying some convenient properties. We will see these "gadgets" used time and again in the construction of polynomial-time reductions.

⋆

Algorithm 14.9 — $3SAT$ to IS reduction.

Input: $3SAT$ formula φ with n variables and m clauses.
Output: Graph $G = (V, E)$ and number k, such that G has an independent set of size k iff φ has a satisfying assignment.

That is, $3SAT(\varphi) = ISET(G, k)$,
Initialize $V \leftarrow \emptyset, E \leftarrow \emptyset$
for every clause $C = y \vee y' \vee y''$ of φ **do**
    Add three vertices $(C, y), (C, y'), (C, y'')$ to V
    Add edges $\{(C, y), (C, y')\}, \{(C, y'), (C, y'')\}, \{(C, y''), (C, y)\}$ to E.
end for
for every distinct clauses C, C' in φ **do**
    for every $i \in [n]$ **do**
        if C contains literal x_i and C' contains literal $\overline{x}_i$ **then**
            Add edge $\{(C, x_i), (C, \overline{x}_i)\}$ to E
        end if
    end for
end for
return $(G = (V, E), m)$

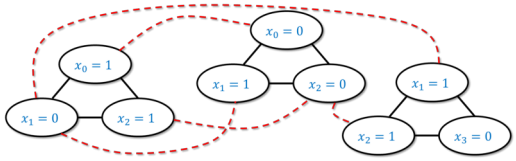


Figure 14.4: An example of the reduction of $3SAT$ to *ISET* for the case the original input formula is $\varphi = (x_0 \vee \overline{x}_1 \vee x_2) \wedge (\overline{x}_0 \vee x_1 \vee \overline{x}_2) \wedge (x_1 \vee x_2 \vee \overline{x}_3)$. We map each clause of φ to a triangle of three vertices, each tagged above with "$x_i = 0$" or "$x_i = 1$" depending on the value of x_i that would satisfy the particular literal. We put an edge between every two literals that are *conflicting* (i.e., tagged with "$x_i = 0$" and "$x_i = 1$" respectively).

Proof of Theorem 14.8. Given a 3SAT formula φ on n variables and with m clauses, we will create a graph G with $3m$ vertices as follows. (See Algorithm 14.9, see also Fig. 14.4 for an example and Fig. 14.5 for Python code.)

- A clause C in φ has the form $C = y \vee y' \vee y''$ where y, y', y'' are *literals* (variables or their negation). For each such clause C, we will add three vertices to G, and label them (C, y), (C, y'), and (C, y'') respectively. We will also add the three edges between all pairs of these vertices, so they form a *triangle*. Since there are m clauses in φ, the graph G will have $3m$ vertices.

- In addition to the above edges, we also add an edge between every pair of vertices of the form (C, y) and (C', y') where y and y' are *conflicting* literals. That is, we add an edge between (C, y) and (C', y') if there is an i such that $y = x_i$ and $y' = \overline{x}_i$ or vice versa.

The algorithm constructing G based on φ takes polynomial time since it involves two loops, the first taking $O(m)$ steps and the second taking $O(m^2 n)$ steps (see Algorithm 14.9). Hence to prove the theorem we need to show that φ is satisfiable if and only if G contains an independent set of m vertices. We now show both directions of this equivalence:

Part 1: Completeness. The "completeness" direction is to show that if φ has a satisfying assignment x^*, then G has an independent set S^* of m vertices. Let us now show this.

Indeed, suppose that φ has a satisfying assignment $x^* \in \{0,1\}^n$. Then for every clause $C = y \vee y' \vee y''$ of φ, one of the literals y, y', y'' must evaluate to *true* under the assignment x^* (as otherwise it would not satisfy φ). We let S be a set of m vertices that is obtained by choosing for every clause C one vertex of the form (C, y) such that y evaluates to true under x^*. (If there is more than one such vertex for the same C, we arbitrarily choose one of them.)

We claim that S is an independent set. Indeed, suppose otherwise that there was a pair of vertices (C, y) and (C', y') in S that have an edge between them. Since we picked one vertex out of each triangle corresponding to a clause, it must be that $C \neq C'$. Hence the only way that there is an edge between (C, y) and (C', y') is if y and y' are conflicting literals (i.e. $y = x_i$ and $y' = \overline{x}_i$ for some i). But then they can't both evaluate to *true* under the assignment x^*, which contradicts the way we constructed the set S. This completes the proof of the completeness condition.

Part 2: Soundness. The "soundness" direction is to show that if G has an independent set S^* of m vertices, then φ has a satisfying assignment $x^* \in \{0,1\}^n$. Let us now show this.

Indeed, suppose that G has an independent set S^* with m vertices. We will define an assignment $x^* \in \{0,1\}^n$ for the variables of φ as follows. For every $i \in [n]$, we set x^*_i according to the following rules:

- If S^* contains a vertex of the form (C, x_i) then we set $x^*_i = 1$.
- If S^* contains a vertex of the form $(C, \overline{x_i})$ then we set $x^*_i = 0$.
- If S^* does not contain a vertex of either of these forms, then it does not matter which value we give to x^*_i, but for concreteness we'll set $x^*_i = 0$.

The first observation is that x^* is indeed well defined, in the sense that the rules above do not conflict with one another, and ask to set x^*_i to be both 0 and 1. This follows from the fact that S^* is an *independent set* and hence if it contains a vertex of the form (C, x_i) then it cannot contain a vertex of the form $(C', \overline{x_i})$.

We now claim that x^* is a satisfying assignment for φ. Indeed, since S^* is an independent set, it cannot have more than one vertex inside each one of the m triangles $(C, y), (C, y'), (C, y'')$ corresponding to a clause of φ. Hence since $|S^*| = m$, it must have exactly one vertex in each such triangle. For every clause C of φ, if (C, y) is the vertex in S^* in the triangle corresponding to C, then by the way we defined x^*, the literal y must evaluate to *true*, which means that x^* satisfies this clause. Therefore x^* satisfies all clauses of φ, which is the definition of a satisfying assignment.

This completes the proof of Theorem 14.8

■

Figure 14.5: The reduction of 3SAT to Independent Set. On the right-hand side is *Python* code that implements this reduction. On the left-hand side is a sample output of the reduction. We use black for the "triangle edges" and red for the "conflict edges". Note that the satisfying assignment $x^* = 0110$ corresponds to the independent set $(0, \neg x_3), (1, \neg x_0), (2, x_2)$.

14.6 SOME EXERCISES AND ANATOMY OF A REDUCTION.

Reductions can be confusing and working out exercises is a great way to gain more comfort with them. Here is one such example. As usual, I recommend you try it out yourself before looking at the solution.

Solved Exercise 14.3 — Vertex cover. A *vertex cover* in a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that every edge touches at least one vertex of S (see Fig. 14.6). The *vertex cover problem* is the task to determine, given a graph G and a number k, whether there exists a vertex cover in the graph with at most k vertices. Formally, this is the function $VC : \{0,1\}^* \to \{0,1\}$ such that for every $G = (V, E)$ and $k \in \mathbb{N}$, $VC(G, k) = 1$ if and only if there exists a vertex cover $S \subseteq V$ such that $|S| \leq k$.

Prove that $3SAT \leq_p VC$.

Solution:

The key observation is that if $S \subseteq V$ is a vertex cover that touches all vertices, then there is no edge e such that both e's endpoints are in the set $\overline{S} = V \setminus S$, and vice versa. In other words, S is a vertex cover if and only if $\overline{S}$ is an independent set. Since the size of $\overline{S}$ is $|V| - |S|$, we see that the polynomial-time map $R(G, k) = (G, n - k)$ (where n is the number of vertices of G) satisfies that $VC(R(G, k)) = ISET(G, k)$ which means that it is a reduction from independent set to vertex cover.

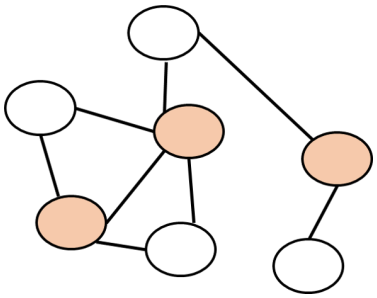


Figure 14.6: A *vertex cover* in a graph is a subset of vertices that touches all edges. In this 7-vertex graph, the 3 filled vertices are a vertex cover.

Solved Exercise 14.4 — Clique is equivalent to independent set. The maximum clique problem corresponds to the function $CLIQUE : \{0,1\}^* \to \{0,1\}$ such that for a graph G and a number k, $CLIQUE(G, k) = 1$ iff there is a subset S of k vertices such that for *every* distinct $u, v \in S$, the edge u, v is in G. Such a set is known as a *clique*.

Prove that $CLIQUE \leq_p ISET$ and $ISET \leq_p CLIQUE$.

Solution:

If $G = (V, E)$ is a graph, we denote by $\overline{G}$ its *complement* which is the graph on the same vertices V and such that for every distinct $u, v \in V$, the edge $\{u, v\}$ is present in $\overline{G}$ if and only if this edge is *not* present in G.

This means that for every set S, S is an independent set in G if and only if S is a *clique* in $\overline{G}$. Therefore for every k, $ISET(G, k) = CLIQUE(\overline{G}, k)$. Since the map $G \mapsto \overline{G}$ can be computed efficiently, this yields a reduction $ISET \leq_p CLIQUE$. Moreover, since $\overline{\overline{G}} = G$ this yields a reduction in the other direction as well.

14.6.1 Dominating set

In the two examples above, the reduction was almost "trivial": the reduction from independent set to vertex cover merely changes the

number k to $n - k$, and the reduction from independent set to clique flips edges to non-edges and vice versa. The following exercise requires a somewhat more interesting reduction.

Solved Exercise 14.5 — Dominating set. A *dominating set* in a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that every $u \in V \setminus S$ is a neighbor in G of some $s \in S$ (see Fig. 14.7). The *dominating set problem* is the task, given a graph $G = (V, E)$ and number k, of determining whether there exists a dominating set $S \subseteq V$ with $|S| \leq k$. Formally, this is the function $DS : \{0,1\}^* \to \{0,1\}$ such that $DS(G, k) = 1$ iff there is a dominating set in G of at most k vertices.

Prove that $ISET \leq_p DS$.

Solution:

Since we know that $ISET \leq_p VC$, using transitivity, it is enough to show that $VC \leq_p DS$. As Fig. 14.7 shows, a dominating set is not the same thing as a vertex cover. However, we can still relate the two problems. The idea is to map a graph G into a graph H such that a vertex cover in G would translate into a dominating set in H and vice versa. We do so by including in H all the vertices and edges of G, but for every edge $\{u, v\}$ of G we also add to H a new vertex $w_{u,v}$ and connect it to both u and v. Let ℓ be the number of isolated vertices in G. The idea behind the proof is that we can transform a vertex cover S of k vertices in G into a dominating set of $k + \ell$ vertices in H by adding to S all the isolated vertices, and moreover we can transform every $k + \ell$-sized dominating set in H into a vertex cover in G. We now give the details.

Description of the algorithm. Given an instance (G, k) for the vertex cover problem, we will map G into an instance (H, k') for the dominating set problem as follows (see Fig. 14.8 for Python implementation):

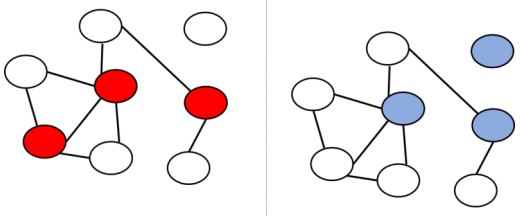


Figure 14.7: A dominating set is a subset S of vertices such that every vertex in the graph is either in S or a neighbor of S. The figure above are two copies of the same graph. The red vertices on the left are a vertex cover that is not a dominating set. The blue vertices on the right are a dominating set that is not a vertex cover.

Algorithm 14.10 — VC to DS reduction.

Input: Graph $G = (V, E)$ and number k.

Output: Graph $H = (V', E')$ and number k', such that G has a vertex cover of size k iff H has a dominating set of size k', that is, $DS(H, k') = VC(G, k)$.

1: Initialize $V' \leftarrow V, E' \leftarrow E$
2: **for** every edge $\{u, v\} \in E$ **do**
3: Add vertex $w_{u,v}$ to V'
4: Add edges $\{u, w_{u,v}\}$, $\{v, w_{u,v}\}$ to E'.
5: **end for**
6: Let $\ell \leftarrow$ number of isolated vertices in G
7: **return** $(H = (V', E')\ ,\ k + \ell)$

Algorithm 14.10 runs in polynomial time, since the loop takes $O(m)$ steps where m is the number of edges, with each step can be implemented in constant or at most linear time (depending on the representation of the graph H). Counting the number of isolated vertices in an n vertex graph G can be done in time $O(n^2)$ if G is represented in the adjacency matrix representation and $O(n)$ time if it is represented in the adjacency list representation. Regardless the algorithm runs in polynomial time.

To complete the proof we need to prove that for every G, k, if H, k' is the output of Algorithm 14.10 on input (G, k), then $DS(H, k') = VC(G, k)$. We split the proof into two parts. The *completeness* part is that if $VC(G, k) = 1$ then $DS(H, k') = 1$. The *soundness* part is that if $DS(H, k') = 1$ then $VC(G, k) = 1$.

Completeness. Suppose that $VC(G, k) = 1$. Then there is a vertex cover $S \subseteq V$ of at most k vertices. Let I be the set of isolated vertices in G and ℓ be their number. Then $|S \cup I| \leq k + \ell$. We claim that $S \cup I$ is a dominating set in H. Indeed for every vertex v of H there are three cases:

- **Case 1:** v is an isolated vertex of G. In this case v is in $S \cup I$.
- **Case 2:** v is a non-isolated vertex of G and hence there is an edge $\{u, v\}$ of G for some u. In this case since S is a vertex cover, one of u, v has to be in S, and hence either v or a neighbor of v has to be in $S \subseteq S \cup I$.
- **Case 3:** v is of the form $w_{u,u'}$ for some two neighbors u, u' in G. But then since S is a vertex cover, one of u, u' has to be in S and hence S contains a neighbor of v.

We conclude that $S \cup I$ is a dominating set of size at most $k' = k+\ell$ in H' and hence under the assumption that $VC(G,k) = 1$, $DS(H',k') = 1$.

Soundness. Suppose that $DS(H,k') = 1$. Then there is a dominating set D of size at most $k' = k+\ell$ in H. For every edge $\{u,v\}$ in the graph G, if D contains the vertex $w_{u,v}$ then we remove this vertex and add u in its place. The only two neighbors of $w_{u,v}$ are u and v, and since u is a neighbor of both $w_{u,v}$ and of v, replacing $w_{u,v}$ with u maintains the property that it is a dominating set. Moreover, this change cannot increase the size of D. Thus following this modification, we can assume that D is a dominating set of at most $k+\ell$ vertices that does not contain any vertices of the form $w_{u,v}$.

Let I be the set of isolated vertices in G. These vertices are also isolated in H and hence must be included in D (an isolated vertex must be in any dominating set, since it has no neighbors). We let $S = D \setminus I$. Then $|S| \leq I$. We claim that S is a vertex cover in G. Indeed, for every edge $\{u,v\}$ of G, either the vertex $w_{u,v}$ or one of its neighbors must be in S by the dominating set property. But since we ensured S doesn't contain any of the vertices of the form $w_{u,v}$, it must be the case that either u or v is in S. This shows that S is a vertex cover of G of size at most k, hence proving that $VC(G,k) = 1$.

A corollary of Algorithm 14.10 and the other reduction we have seen so far is that if $DS \in \mathbf{P}$ (i.e., dominating set has a polynomial-time algorithm) then $3SAT \in \mathbf{P}$ (i.e., $3SAT$ has a polynomial-time algorithm). By the contra-positive, if $3SAT$ does *not* have a polynomial-time algorithm then neither does dominating set.

Figure 14.8: Python implementation of the reduction from vertex cover to dominating set, together with an example of an input graph and the resulting output graph. This reduction allows to transform a hypothetical polynomial-time algorithm for dominating set (a "whistling pig") into a hypothetical polynomial-time algorithm for vertex-cover (a "flying horse").

14.6.2 Anatomy of a reduction

The reduction of Solved Exercise 14.5 gives a good illustration of the anatomy of a reduction. A reduction consists of four parts:

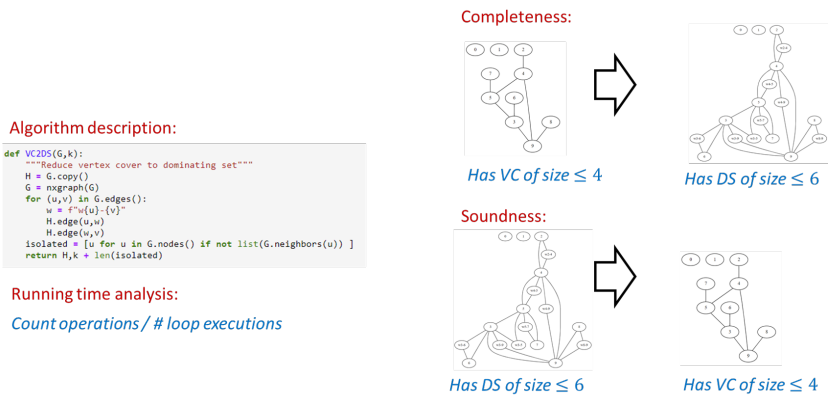


Figure 14.9: The four components of a reduction, illustrated for the particular reduction of vertex cover to dominating set. A reduction from problem F to problem G is an algorithm that maps an input x for F into an input $R(x)$ for G. To show that the reduction is correct we need to show the properties of *efficiency*: algorithm R runs in polynomial time, *completeness*: if $F(x) = 1$ then $G(R(x)) = 1$, and *soundness*: if $F(R(x)) = 1$ then $G(x) = 1$.

- **Algorithm description:** This is the description of *how* the algorithm maps an input into the output. For example, in Solved Exercise 14.5 this is the description of how we map an instance (G, k) of the *vertex cover* problem into an instance (H, k') of the *dominating set* problem.

- **Algorithm analysis:** It is not enough to describe *how* the algorithm works but we need to also explain *why* it works. In particular we need to provide an *analysis* explaining why the reduction is both *efficient* (i.e., runs in polynomial time) and *correct* (satisfies that $G(R(x)) = F(x)$ for every x). Specifically, the components of analysis of a reduction R include:

  - **Efficiency:** We need to show that R runs in polynomial time. In most reductions we encounter this part is straightforward, as the reductions we typically use involve a constant number of nested loops, each involving a constant number of operations. For example, the reduction of Solved Exercise 14.5 just enumerates over the edges and vertices of the input graph.
  - **Completeness:** In a reduction R demonstrating $F \leq_p G$, the *completeness* condition is the condition that for every $x \in \{0,1\}^*$, if $F(x) = 1$ then $G(R(x)) = 1$. Typically we construct the reduction to ensure that this holds, by giving a way to map a "certificate/solution" certifying that $F(x) = 1$ into a solution certifying that $G(R(x)) = 1$. For example, in Solved Exercise 14.5 we constructed the graph H such that for every vertex cover S in G, the set $S \cup I$ (where I is the isolated vertices) would be a dominating set in H.
  - **Soundness:** This is the condition that if $F(x) = 0$ then $G(R(x)) = 0$ or (taking the contrapositive) if $G(R(x)) = 1$ then $F(x) = 1$. This is sometimes straightforward but can often be

harder to show than the completeness condition, and in more advanced reductions (such as the reduction $3SAT \leq_p ISET$ of Theorem 14.8) demonstrating soundness is the main part of the analysis. For example, in Solved Exercise 14.5 to show soundness we needed to show that for *every* dominating set D in the graph H, there exists a vertex cover S of size at most $|D| - \ell$ in the graph G (where ℓ is the number of isolated vertices). This was challenging since the dominating set D might not be necessarily the one we "had in mind". In particular, in the proof above we needed to modify D to ensure that it does not contain vertices of the form $w_{u,v}$, and it was important to show that this modification still maintains the property that D is a dominating set, and also does not make it bigger.

Whenever you need to provide a reduction, you should make sure that your description has all these components. While it is sometimes tempting to weave together the description of the reduction and its analysis, it is usually clearer if you separate the two, and also break down the analysis to its three components of efficiency, completeness, and soundness.

14.7 REDUCING INDEPENDENT SET TO MAXIMUM CUT

We now show that the independent set problem reduces to the *maximum cut* (or "max cut") problem, modeled as the function $MAXCUT$ that on input a pair (G, k) outputs 1 iff G contains a cut of at least k edges. Since both are graph problems, a reduction from independent set to max cut maps one graph into the other, but as we will see the output graph does not have to have the same vertices or edges as the input graph.

Theorem 14.11 — Hardness of Max Cut. $ISET \leq_p MAXCUT$

Proof Idea:

We will map a graph G into a graph H such that a large independent set in G becomes a partition cutting many edges in H. We can think of a cut in H as coloring each vertex either "blue" or "red". We will add a special "source" vertex s^*, connect it to all other vertices, and assume without loss of generality that it is colored blue. Hence the more vertices we color red, the more edges from s^* we cut. Now, for every edge u, v in the original graph G we will add a special "gadget" which will be a small subgraph that involves u,v, the source s^*, and two other additional vertices. We design the gadget in a way so that if the red vertices are not an independent set in G then the corresponding cut in H will be "penalized" in the sense that it would

not cut as many edges. Once we set for ourselves this objective, it is not hard to find a gadget that achieves it— see the proof below. Once again the **takeaway technique** is to use (this time a slightly more clever) gadget.

⋆

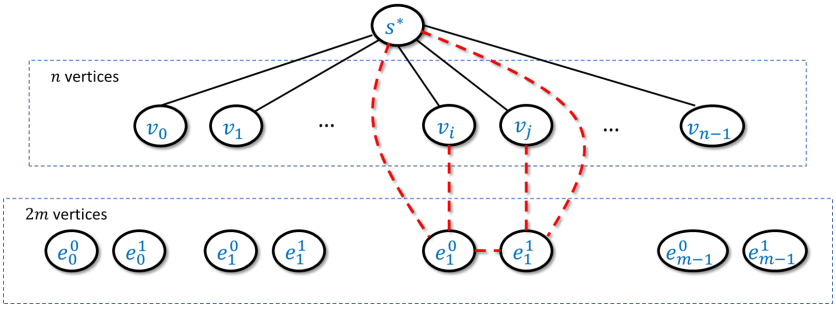


Figure 14.10: In the reduction of *ISET* to *MAXCUT* we map an n-vertex m-edge graph G into the $n + 2m + 1$ vertex and $n + 5m$ edge graph H as follows. The graph H contains a special "source" vertex s^*,n vertices $v_0, \ldots, v_{n-1}$, and $2m$ vertices $e_0^0, e_0^1, \ldots, e_{m-1}^0, e_{m-1}^1$ with each pair corresponding to an edge of G. We put an edge between s^* and v_i for every $i \in [n]$, and if the t-th edge of G was (v_i, v_j) then we add the five edges $(s^*, e_t^0), (s^*, e_t^1), (v_i, e_t^0), (v_j, e_t^1), (e_t^0, e_t^1)$. The intent is that if we cut at most one of v_i, v_j from s^* then we'll be able to cut 4 out of these five edges, while if we cut both v_i and v_j from s^* then we'll be able to cut at most three of them.

Proof of Theorem 14.11. We will transform a graph G of n vertices and m edges into a graph H of $n + 1 + 2m$ vertices and $n + 5m$ edges in the following way (see also Fig. 14.10). The graph H contains all vertices of G (though not the edges between them!) and in addition H also has:

* A special vertex s^* that is connected to all the vertices of G

* For every edge $e = \{u, v\} \in E(G)$, two vertices e_0, e_1 such that e_0 is connected to u and e_1 is connected to v, and moreover we add the edges $\{e_0, e_1\}, \{e_0, s^*\}, \{e_1, s^*\}$ to H.

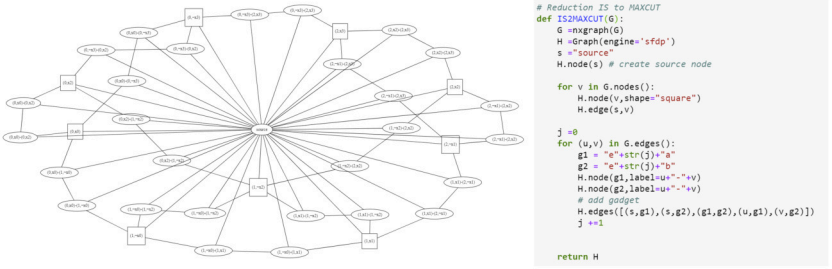
Theorem 14.11 will follow by showing that G contains an independent set of size at least k if and only if H has a cut cutting at least $k + 4m$ edges. We now prove both directions of this equivalence:

Part 1: Completeness. If I is an independent k-sized set in G, then we can define S to be a cut in H of the following form: we let S contain all the vertices of I and for every edge $e = \{u, v\} \in E(G)$, if $u \in I$ and $v \notin I$ then we add e_1 to S; if $u \notin I$ and $v \in I$ then we add e_0 to S; and if $u \notin I$ and $v \notin I$ then we add both e_0 and e_1 to S. (We don't need to worry about the case that both u and v are in I since it is an independent set.) We can verify that in all cases the number of edges from S to its complement in the gadget corresponding to e will be four (see Fig. 14.11). Since s^* is not in S, we also have k edges from s^* to I, for a total of $k + 4m$ edges.

Part 2: Soundness. Suppose that S is a cut in H that cuts at least $C = k + 4m$ edges. We can assume that s^* is not in S (otherwise we can "flip" S to its complement $\overline{S}$, since this does not change the size of the cut). Now let I be the set of vertices in S that correspond to the original vertices of G. If I was an independent set of size k then we

would be done. This might not always be the case but we will see that if I is not an independent set then it's also larger than k. Specifically, we define $m_{in} = |E(I,I)|$ be the set of edges in G that are contained in I and let $m_{out} = m - m_{in}$ (i.e., if I is an independent set then $m_{in} = 0$ and $m_{out} = m$). By the properties of our gadget we know that for every edge $\{u,v\}$ of G, we can cut at most three edges when both u and v are in S, and at most four edges otherwise. Hence the number C of edges cut by S satisfies $C \leq |I| + 3m_{in} + 4m_{out} = |I| + 3m_{in} + 4(m - m_{in}) = |I| + 4m - m_{in}$. Since $C = k + 4m$ we get that $|I| - m_{in} \geq k$. Now we can transform I into an independent set I' by going over every one of the m_{in} edges that are inside I and removing one of the endpoints of the edge from it. The resulting set I' is an independent set in the graph G of size $|I| - m_{in} \geq k$ and so this concludes the proof of the soundness condition.

■



```
# Reduction IS to MAXCUT
def IS2MAXCUT(G):
    G =nxgraph(G)
    H =Graph(engine='sfdp')
    s ="source"
    H.node(s) # create source node

    for v in G.nodes():
        H.node(v,shape="square")
        H.edge(s,v)

    j =0
    for (u,v) in G.edges():
        g1 = "e"+str(j)+"a"
        g2 = "e"+str(j)+"b"
        H.node(g1,label=u+"-"+v)
        H.node(g2,label=u+"-"+v)
        # add gadget
        H.edges([(s,g1),(s,g2),(g1,g2),(u,g1),(v,g2)])
        j +=1


    return H
```

Figure 14.12: The reduction of independent set to max cut. On the right-hand side is Python code implementing the reduction. On the left-hand side is an example output of the reduction where we apply it to the independent set instance that is obtained by running the reduction of Theorem 14.8 on the 3CNF formula $(x_0 \vee \overline{x}_3 \vee x_3) \wedge (\overline{x}_0 \vee x_1 \vee \overline{x}_2) \wedge (x_1 \vee x_2 \vee x_3)$. by enumerating all possibilities one can verify that if both u and v are blue, then no matter how we color the intermediate vertices e_t^0, e_t^1, we will cut at most three edges from the gadget. The figure above contains only the gadget edges and ignores the edges connecting s^* to the vertices $v_0, \ldots, v_{n-1}$.

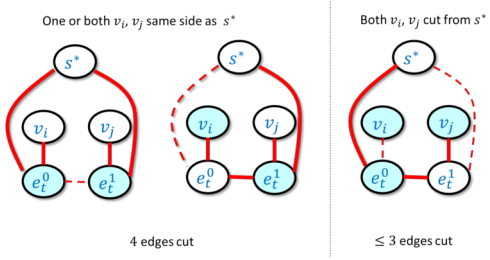


Figure 14.11: In the reduction of independent set to max cut, for every $t \in [m]$, we have a "gadget" corresponding to the t-th edge $e_t = \{v_i, v_j\}$ in the original graph. If we think of the side of the cut containing the special source vertex s^* as "white", and the other side as "blue", then the leftmost and center figures show that if v_i and v_j are not both blue then we can cut four edges from the gadget. In contrast, if both u and v are blue, then no matter how we color the intermediate vertices e_t^0, e_t^1, we will cut at most three edges from the gadget. The figure above contains only the gadget edges and ignores the edges connecting s^* to the vertices $v_0, \ldots, v_{n-1}$.

14.8 REDUCING 3SAT TO LONGEST PATH

Note: This section is still a little messy; feel free to skip it or just read it without going into the proof details. The proof appears in Section 7.5 in Sipser's book.

One of the most basic algorithms in Computer Science is Dijkstra's algorithm to find the *shortest path* between two vertices. We now show that in contrast, an efficient algorithm for the *longest path* problem would imply a polynomial-time algorithm for 3SAT.

Theorem 14.12 — Hardness of longest path.

$$3SAT \leq_p LONGPATH$$

Proof Idea:

To prove Theorem 14.12 need to show how to transform a 3CNF formula φ into a graph G and two vertices s, t such that G has a path

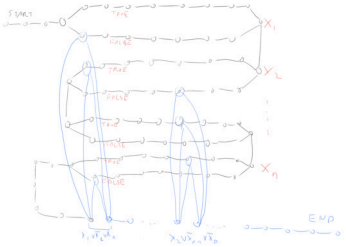


Figure 14.13: We can transform a 3SAT formula φ into a graph G such that the longest path in the graph G would correspond to a satisfying assignment in φ. In this graph, the black colored part corresponds to the variables of φ and the blue colored part corresponds to the vertices. A sufficiently long path would have to first "snake" through the black part, for each variable choosing either the "upper path" (corresponding to assigning it the value `True`) or the "lower path" (corresponding to assigning it the value `False`). Then to achieve maximum length the path would traverse through the blue part, where to go between two vertices corresponding to a clause such as $x_{17} \vee \overline{x}_{32} \vee x_{57}$, the corresponding vertices would have to have been not traversed before.

of length at least k if and only if φ is satisfiable. The idea of the reduction is sketched in Fig. 14.13 and Fig. 14.14. We will construct a graph that contains a potentially long "snaking path" that corresponds to all variables in the formula. We will add a "gadget" corresponding to each clause of φ in a way that we would only be able to use the gadgets if we have a satisfying assignment.

⋆

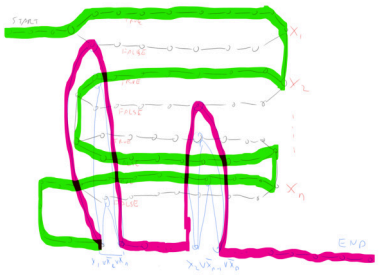


Figure 14.14: The graph above with the longest path marked on it, the part of the path corresponding to variables is in green and part corresponding to the clauses is in pink.

```python
def TSAT2LONGPATH(ϕ):
    """Reduce 3SAT to LONGPATH"""
    def var(v): # return variable and True/False depending
     ↪  if positive or negated
        return int(v[2:]),False if v[0]=="¬" else
         ↪  int(v[1:]),True
    n = numvars(ϕ)
    clauses = getclauses(ϕ)
    m = len(clauses)
    G =Graph()
    G.edge("start","start_0")
    for i in range(n): # add 2 length-m paths per variable
        G.edge(f"start_{i}",f"v_{i}_{0}_T")
        G.edge(f"start_{i}",f"v_{i}_{0}_F")
        for j in range(m-1):
            G.edge(f"v_{i}_{j}_T",f"v_{i}_{j+1}_T")
            G.edge(f"v_{i}_{j}_F",f"v_{i}_{j+1}_F")
        G.edge(f"v_{i}_{m-1}_T",f"end_{i}")
        G.edge(f"v_{i}_{m-1}_F",f"end_{i}")
        if i<n-1:
            G.edge(f"end_{i}",f"start_{i+1}")
    G.edge(f"end_{n-1}","start_clauses")
    for j,C in enumerate(clauses): # add gadget for each
     ↪  clause
        for v in enumerate(C):
            i,sign = var(v[1])
            s = "F" if sign else "T"
            G.edge(f"C_{j}_in",f"v_{i}_{j}_{s}")
            G.edge(f"v_{i}_{j}_{s}",f"C_{j}_out")
        if j<m-1:
            G.edge(f"C_{j}_out",f"C_{j+1}_in")
    G.edge("start_clauses","C_0_in")
    G.edge(f"C_{m-1}_out","end")
    return G, 1+n*(m+1)+1+2*m+1
```

Proof of Theorem 14.12. We build a graph G that "snakes" from s to t as follows. After s we add a sequence of n long loops. Each loop has an

"upper path" and a "lower path". A simple path cannot take both the upper path and the lower path, and so it will need to take exactly one of them to reach s from t.

Our intention is that a path in the graph will correspond to an assignment $x \in \{0,1\}^n$ in the sense that taking the upper path in the ith loop corresponds to assigning $x_i = 1$ and taking the lower path corresponds to assigning $x_i = 0$. When we are done snaking through all the n loops corresponding to the variables to reach t we need to pass through m "obstacles": for each clause j we will have a small gadget consisting of a pair of vertices s_j, t_j that have three paths between them. For example, if the jth clause had the form $x_{17} \vee \overline{x}_{55} \vee x_{72}$ then one path would go through a vertex in the lower loop corresponding to x_{17}, one path would go through a vertex in the upper loop corresponding to x_{55} and the third would go through the lower loop corresponding to x_{72}. We see that if we went in the first stage according to a satisfying assignment then we will be able to find a free vertex to travel from s_j to t_j. We link t_1 to s_2, t_2 to s_3, etc and link t_m to t. Thus a satisfying assignment would correspond to a path from s to t that goes through one path in each loop corresponding to the variables, and one path in each loop corresponding to the clauses. We can make the loop corresponding to the variables long enough so that we must take the entire path in each loop in order to have a fighting chance of getting a path as long as the one corresponds to a satisfying assignment. But if we do that, then the only way if we are able to reach t is if the paths we took corresponded to a satisfying assignment, since otherwise we will have one clause j where we cannot reach t_j from s_j without using a vertex we already used before.

■

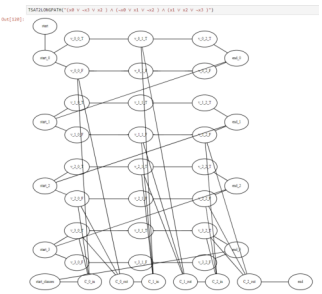


Figure 14.15: The result of applying the reduction of $3SAT$ to *LONGPATH* to the formula $(x_0 \vee \neg x_3 \vee x_2) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee \neg x_3)$.

14.8.1 Summary of relations

We have shown that there are a number of functions F for which we can prove a statement of the form "If $F \in \mathbf{P}$ then $3SAT \in \mathbf{P}$". Hence coming up with a polynomial-time algorithm for even one of these problems will entail a polynomial-time algorithm for $3SAT$ (see for example Fig. 14.16). In Chapter 15 we will show the inverse direction ("If $3SAT \in \mathbf{P}$ then $F \in \mathbf{P}$") for these functions, hence allowing us to conclude that they have *equivalent complexity* to $3SAT$.

Chapter Recap

- The computational complexity of many seemingly unrelated computational problems can be related to one another through the use of *reductions*.

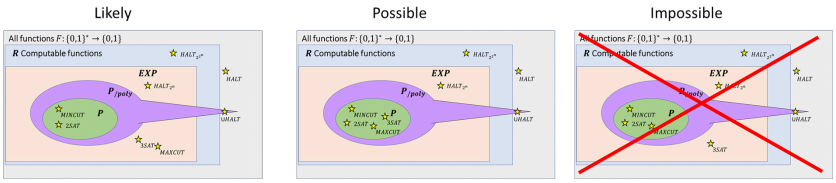


Figure 14.16: So far we have shown that $\mathbf{P} \subseteq \mathbf{EXP}$ and that several problems we care about such as $3SAT$ and $MAXCUT$ are in **EXP** but it is not known whether or not they are in **P**. However, since $3SAT \leq_p MAXCUT$ we can rule out the possiblity that $MAXCUT \in \mathbf{P}$ but $3SAT \notin \mathbf{P}$. The relation of $\mathbf{P}_{/\mathbf{poly}}$ to the class **EXP** is not known. We know that **EXP** does not contain $\mathbf{P}_{/\mathbf{poly}}$ since the latter even contains uncomputable functions, but we do not know whether ot not $\mathbf{EXP} \subseteq \mathbf{P}_{/\mathbf{poly}}$ (though it is believed that this is not the case and in particular that both $3SAT$ and $MAXCUT$ are not in $\mathbf{P}_{/\mathbf{poly}}$).

- If $F \leq_p G$ then a polynomial-time algorithm for G can be transformed into a polynomial-time algorithm for F.
- Equivalently, if $F \leq_p G$ and F does *not* have a polynomial-time algorithm then neither does G.
- We've developed many techniques to show that $3SAT \leq_p F$ for interesting functions F. Sometimes we can do so by using *transitivity* of reductions: if $3SAT \leq_p G$ and $G \leq_p F$ then $3SAT \leq_p F$.

14.9 EXERCISES

14.10 BIBLIOGRAPHICAL NOTES

Several notions of reductions are defined in the literature. The notion defined in Definition 14.1 is often known as a *mapping reduction*, *many to one reduction* or a *Karp reduction*.

The *maximal* (as opposed to *maximum*) independent set is the task of finding a "local maximum" of an independent set: an independent set S such that one cannot add a vertex to it without losing the independence property (such a set is known as a *vertex cover*). Unlike finding a *maximum* independent set, finding a *maximal* independent set can be done efficiently by a greedy algorithm, but this local maximum can be much smaller than the global maximum.

Reduction of independent set to max cut taken from these notes. Image of Hamiltonian Path through Dodecahedron by Christoph Sommer.

We have mentioned that the line between reductions used for algorithm design and showing hardness is sometimes blurry. An excellent example for this is the area of *SAT Solvers* (see [Gom+08]). In this field people use algorithms for SAT (that take exponential time in the worst case but often are much faster on many instances in practice) together with reductions of the form $F \leq_p SAT$ to derive algorithms for other functions F of interest.

Learning Objectives:

- **Introduce the class NP capturing a great many important computational problems**
- **NP-completeness: evidence that a problem might be intractable.**
- **The P vs NP problem.**

15
NP, NP completeness, and the Cook-Levin Theorem

> *"In this paper we give theorems that suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually"*, Richard Karp, 1972

> *"Sad to say, but it will be many more years, if ever before we really understand the Mystical Power of Twoness... 2-SAT is easy, 3-SAT is hard, 2-dimensional matching is easy, 3-dimensional matching is hard. Why? oh, Why?"* Eugene Lawler

So far we have shown that 3SAT is no harder than Quadratic Equations, Independent Set, Maximum Cut, and Longest Path. But to show that these problems are *computationally equivalent* we need to give reductions in the other direction, reducing each one of these problems to 3SAT as well. It turns out we can reduce all three problems to 3SAT in one fell swoop.

In fact, this result extends far beyond these particular problems. All of the problems we discussed in Chapter 14, and a great many other problems, share the same commonality: they are all *search* problems, where the goal is to decide, given an instance x, whether there exists a *solution* y that satisfies some condition that can be verified in polynomial time. For example, in 3SAT, the instance is a formula and the solution is an assignment to the variable; in Max-Cut the instance is a graph and the solution is a cut in the graph; and so on and so forth. It turns out that *every* such search problem can be reduced to 3SAT.

This chapter: A non-mathy overview

In this chapter we will see the definition of the complexity class **NP**- one of the most important definitions in this book, and the Cook-Levin Theorem- one of the most important theorems in it. Intuitively, the class **NP** corresponds to the class of problems where it is *easy to verify* a solution (i.e., verification can be done by a polynomial-time algorithm). For example, finding a satistfying assignment to a 2SAT or

3SAT formula is such a problem, since if we are given an assignment to the variables a 2SAT or 3SAT formula then we can efficiently verify that it satisfies all constraints. More precisely, **NP** is the class of decision problems (i.e., Boolean functions or languages) corresponding to determining the existence of such a solution, though we will see in Chapter 16 that the decision and search problems are closely related. As the examples of 2SAT and 3SAT show, there are some computational problems (i.e., functions) in **NP** for which we have a polynomial-time algorithm, and some for which no such algorithm is known. It is an outstanding open question whether or not all functions in **NP** have a polynomial-time algorithm, or in other words (to use just a little bit of math) whether or not **P** $=$ **NP**. In this chapter we will see that there are some functions in **NP** that are in a precise sense "hardest in all of **NP**" in the sense that *if* even one of these functions has a polynomial-time algorithm then *all* functions in **NP** have such an algorithm. Such functions are known as **NP** *complete*. The Cook-Levin Theorem states that 3SAT is **NP** complete. Using a complex web of polynomial-time reductions, researchers have derived from the Cook-Levin theorem the **NP**-completeness of thousands of computational problems from all areas of mathematics, natural and social sciences, engineering, and more. These results provide strong evidence that all of these problems cannot be solved in the worst-case by polynomial-time algorithm.

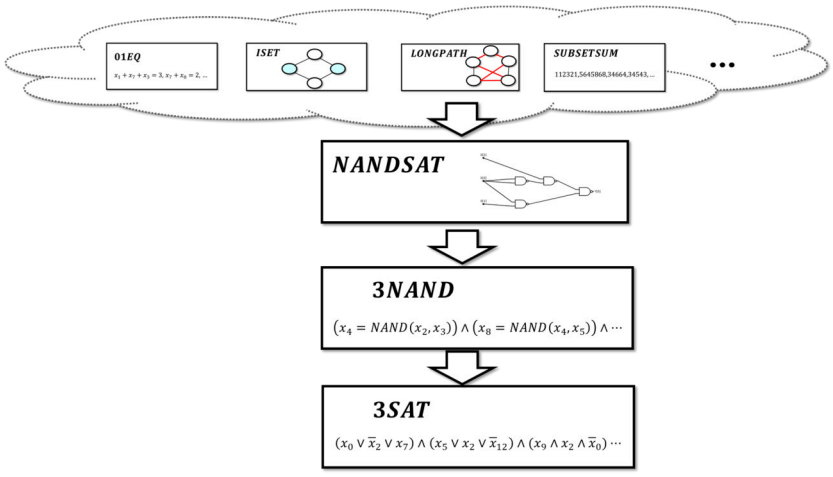


Figure 15.1: Overview of the results of this chapter. We define **NP** to contain all decision problems for which a solution can be efficiently *verified*. The main result of this chapter is the *Cook Levin Theorem* (Theorem 15.6) which states that 3*SAT* has a polynomial-time algorithm if and only if *every* problem in **NP** has a polynomial-time algorithm. Another way to state this theorem is that 3*SAT* is **NP** *complete*. We will prove the Cook-Levin theorem by defining the two intermediate problems *NANDSAT* and 3*NAND*, proving that *NANDSAT* is **NP** complete, and then proving that $NANDSAT \leq_p 3NAND \leq_p 3SAT$.

15.1 THE CLASS NP

To make the above precise, we will make the following mathematical definition. We define the class **NP** to contain all Boolean functions that correspond to a *search problem* of the form above. That is, a Boolean function F is in **NP** if F has the form that on input a string x, $F(x) = 1$ if and only if there exists a "solution" string w such that the pair (x, w) satisfies some polynomial-time checkable condition. Formally, **NP** is defined as follows:

Definition 15.1 — NP. We say that $F : \{0,1\}^* \to \{0,1\}$ is in **NP** if there exists some integer $a > 0$ and $V : \{0,1\}^* \to \{0,1\}$ such that $V \in \mathbf{P}$ and for every $x \in \{0,1\}^n$,

$$F(x) = 1 \Leftrightarrow \exists_{w \in \{0,1\}^{n^a}} \text{ s.t. } V(xw) = 1 \,. \tag{15.1}$$

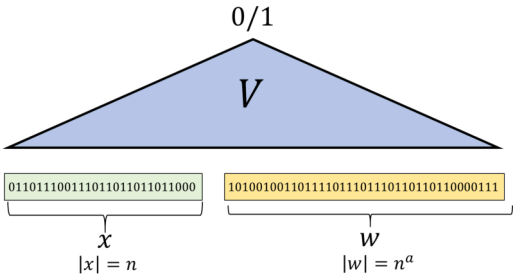


Figure 15.2: The class **NP** corresponds to problems where solutions can be *efficiently verified*. That is, this is the class of functions F such that $F(x) = 1$ if there is a "solution" w of length polynomial in $|x|$ that can be verified by a polynomial-time algorithm V.

In other words, for F to be in **NP**, there needs to exist some polynomial-time computable verification function V, such that if $F(x) = 1$ then there must exist w (of length polynomial in $|x|$) such that $V(xw) = 1$, and if $F(x) = 0$ then for *every* such w, $V(xw) = 0$. Since the existence of this string w certifies that $F(x) = 1$, w is often referred to as a *certificate*, *witness*, or *proof* that $F(x) = 1$.

See also Fig. 15.2 for an illustration of Definition 15.1. The name **NP** stands for "non-deterministic polynomial time" and is used for historical reasons; see the bibiographical notes. The string w in (15.1) is sometimes known as a *solution*, *certificate*, or *witness* for the instance x.

Solved Exercise 15.1 — Alternative definition of NP. Show that the condition that $|w| = |x|^a$ in Definition 15.1 can be replaced by the condition that $|w| \leq p(|x|)$ for some polynomial p. That is, prove that for every $F : \{0,1\}^* \to \{0,1\}$, $F \in$ **NP** if and only if there is a polynomial-time Turing machine V and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$ $F(x) = 1$ if and only if there exists $w \in \{0,1\}^*$ with $|w| \leq p(|x|)$ such that $V(x, w) = 1$.

■

Solution:

The "only if" direction (namely that if $F \in$ **NP** then there is an algorithm V and a polynomial p as above) follows immediately from Definition 15.1 by letting $p(n) = n^a$. For the "if" direction, the idea is that if a string w is of size at most $p(n)$ for degree d polynomial p, then there is some n_0 such that for all $n > n_0$, $|w| < n^{d+1}$. Hence we can encode w by a string of exactly length

n^{d+1} by padding it with 1 and an appropriate number of zeroes. Hence if there is an algorithm V and polynomial p as above, then we can define an algorithm V' that does the following on input x, w' with $|x| = n$ and $|w'| = n^a$:

- If $n \leq n_0$ then $V'(x, w')$ ignores w' and enumerates over all w of length at most $p(n)$ and outputs 1 if there exists w such that $V(x, w) = 1$. (Since $n < n_0$, this only takes a constant number of steps.)
- If $n > n_0$ then $V'(x, w')$ "strips out" the padding by dropping all the rightmost zeroes from w until it reaches out the first 1 (which it drops as well) and obtains a string w. If $|w| \leq p(n)$ then V' outputs $V(x, w)$.

Since V runs in polynomial time, V' runs in polynomial time as well, and by definition for every x, there exists $w' \in \{0,1\}^{|x|^a}$ such that $V'(xw') = 1$ if and only if there exists $w \in \{0,1\}^*$ with $|w| \leq p(|x|)$ such that $V(xw) = 1$.

The definition of **NP** means that for every $F \in$ **NP** and string $x \in \{0,1\}^*$, $F(x) = 1$ if and only if there is a *short and efficiently verifiable proof* of this fact. That is, we can think of the function V in Definition 15.1 as a *verifier* algorithm, similar to what we've seen in Section 11.1. The verifier checks whether a given string $w \in \{0,1\}^*$ is a valid proof for the statement "$F(x) = 1$". Essentially all proof systems considered in mathematics involve line-by-line checks that can be carried out in polynomial time. Thus the heart of **NP** is asking for statements that have *short* (i.e., polynomial in the size of the statements) proofs. Indeed, as we will see in Chapter 16, Kurt Gödel phrased the question of whether **NP** $=$ **P** as asking whether "the mental work of a mathematician [in proving theorems] could be completely replaced by a machine".

R

Remark 15.2 — NP not (necessarily) closed under complement. Definition 15.1 is *asymmetric* in the sense that there is a difference between an output of 1 and an output of 0. You should make sure you understand why this definition does *not* guarantee that if $F \in$ **NP** then the function $1 - F$ (i.e., the map $x \mapsto 1 - F(x)$) is in **NP** as well.

In fact, it is believed that there do exist functions F such that $F \in$ **NP** but $1 - F \notin$ **NP**. For example, as shown below, $3SAT \in$ **NP**, but the function $\overline{3SAT}$ that on input a 3CNF formula φ outputs 1 if and only if φ is *not* satisfiable is not known (nor believed) to be in

NP. This is in contrast to the class **P** which *does* satisfy that if $F \in \mathbf{P}$ then $1 - F$ is in **P** as well.

15.1.1 Examples of functions in NP

We now present some examples of functions that are in the class **NP**. We start with the canonical example of the $3SAT$ function.

■ **Example 15.3 — $3SAT \in \mathbf{NP}$.** $3SAT$ is in **NP** since for every ℓ-variable formula φ, $3SAT(\varphi) = 1$ if and only if there exists a satisfying assignment $x \in \{0,1\}^\ell$ such that $\varphi(x) = 1$, and we can check this condition in polynomial time.

The above reasoning explains why $3SAT$ is in **NP**, but since this is our first example, we will now belabor the point and expand out in full formality the precise representation of the witness w and the algorithm V that demonstrate that $3SAT$ is in **NP**. Since demonstrating that functions are in **NP** is fairly straightforward, in future cases we will not use as much detail, and the reader can also feel free to skip the rest of this example.

Using Solved Exercise 15.1, it is OK if witness is of size at most polynomial in the input length n, rather than of precisely size n^a for some integer $a > 0$. Specifically, we can represent a 3CNF formula φ with k variables and m clauses as a string of length $n = O(m \log k)$, since every one of the m clauses involves three variables and their negation, and the identity of each variable can be represented using $\lceil \log_2 k \rceil$. We assume that every variable participates in some clause (as otherwise it can be ignored) and hence that $m \geq k$, which in particular means that the input length n is at least as large as m and k.

We can represent an assignment to the k variables using a k-length string w. The following algorithm checks whether a given w satisfies the formula φ:

Algorithm 15.4 — Verifier for $3SAT$.

Input: 3CNF formula φ on k variables and with m clauses, string $w \in \{0,1\}^k$

Output: 1 iff w satisfies φ

```
for j ∈ [m] do
    Let ℓ_1 ∨ ℓ_2 ∨ ℓ_j be the j-th clause of φ
    if w violates all three literals then
        return 0
    end if
end for
return 1
```

Algorithm 15.4 takes $O(m)$ time to enumerate over all clauses, and will return 1 if and only if y satisfies all the clauses.

Here are some more examples for problems in **NP**. For each one of these problems we merely sketch how the witness is represented and why it is efficiently checkable, but working out the details can be a good way to get more comfortable with Definition 15.1:

- *QUADEQ* is in **NP** since for every ℓ-variable instance of quadratic equations E, $QUADEQ(E) = 1$ if and only if there exists an assignment $x \in \{0,1\}^\ell$ that satisfies E. We can check the condition that x satisfies E in polynomial time by enumerating over all the equations in E, and for each such equation e, plug in the values of x and verify that e is satisfied.

- *ISET* is in **NP** since for every graph G and integer k, $ISET(G,k) = 1$ if and only if there exists a set S of k vertices that contains no pair of neighbors in G. We can check the condition that S is an independent set of size $\geq k$ in polynomial time by first checking that $|S| \geq k$ and then enumerating over all edges $\{u,v\}$ in G, and for each such edge verify that either $u \notin S$ or $v \notin S$.

- *LONGPATH* is in **NP** since for every graph G and integer k, $LONGPATH(G,k) = 1$ if and only if there exists a simple path P in G that is of length at least k. We can check the condition that P is a simple path of length k in polynomial time by checking that it has the form $(v_0, v_1, \ldots, v_k)$ where each v_i is a vertex in G, no v_i is repeated, and for every $i \in [k]$, the edge $\{v_i, v_{i+1}\}$ is present in the graph.

- *MAXCUT* is in **NP** since for every graph G and integer k, $MAXCUT(G,k) = 1$ if and only if there exists a cut $(S, \overline{S})$ in G that cuts at least k edges. We can check that condition that $(S, \overline{S})$ is a cut of value at least k in polynomial time by checking that S is a

subset of G's vertices and enumerating over all the edges $\{u, v\}$ of G, counting those edges such that $u \in S$ and $v \notin S$ or vice versa.

15.1.2 Basic facts about NP

The definition of **NP** is one of the most important definitions of this book, and is worth while taking the time to digest and internalize. The following solved exercises establish some basic properties of this class. As usual, I highly recommend that you try to work out the solutions yourself.

Solved Exercise 15.2 — Verifying is no harder than solving. Prove that **P** $\subseteq$ **NP**.

Solution:

Suppose that $F \in$ **P**. Define the following function V: $V(x0^n) = 1$ iff $n = |x|$ and $F(x) = 1$. (V outputs 0 on all other inputs.) Since $F \in$ **P** we can clearly compute V in polynomial time as well.

Let $x \in \{0,1\}^n$ be some string. If $F(x) = 1$ then $V(x0^n) = 1$. On the other hand, if $F(x) = 0$ then for every $w \in \{0,1\}^n$, $V(xw) = 0$. Therefore, setting $a = 1$ (i.e. $w \in \{0,1\}^{n^1}$), we see that V satisfies (15.1), and establishes that $F \in$ **NP**.

Remark 15.5 — NP does not mean non-polynomial!. People sometimes think that **NP** stands for "non-polynomial time". As Solved Exercise 15.2 shows, this is far from the truth, and in fact every polynomial-time computable function is in **NP** as well.

If F is in **NP** it certainly does *not* mean that F is hard to compute (though it does not, as far as we know, necessarily mean that it's easy to compute either). Rather, it means that F is *easy to verify*, in the technical sense of Definition 15.1.

Solved Exercise 15.3 — NP is in exponential time. Prove that **NP** $\subseteq$ **EXP**.

Solution:

Suppose that $F \in$ **NP** and let V be the polynomial-time computable function that satisfies (15.1) and a the corresponding constant. Then given every $x \in \{0,1\}^n$, we can check whether $F(x) = 1$ in time $poly(n) \cdot 2^{n^a} = o(2^{n^{a+1}})$ by enumerating over all the 2^{n^a} strings $w \in \{0,1\}^{n^a}$ and checking whether $V(xw) = 1$, in which case we return 1. If $V(xw) = 0$ for every such w then we return 0. By construction, the algorithm above will run in time at

most exponential in its input length and by the definition of **NP** it will return $F(x)$ for every x.

■

Solved Exercise 15.2 and Solved Exercise 15.3 together imply that

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP} \ .$$

The time hierarchy theorem (Theorem 13.9) implies that $\mathbf{P} \subsetneq \mathbf{EXP}$ and hence at least one of the two inclusions $\mathbf{P} \subseteq \mathbf{NP}$ or $\mathbf{NP} \subseteq \mathbf{EXP}$ is *strict*. It is believed that both of them are in fact strict inclusions. That is, it is believed that there are functions in **NP** that cannot be computed in polynomial time (this is the $\mathbf{P} \neq \mathbf{NP}$ conjecture) and that there are functions F in **EXP** for which we cannot even efficiently *certify* that $F(x) = 1$ for a given input x. One function F that is believed to lie in $\mathbf{EXP} \setminus \mathbf{NP}$ is the function $\overline{3SAT}$ defined as $\overline{3SAT}(\varphi) = 1 - 3SAT(\varphi)$ for every 3CNF formula φ. The conjecture that $\overline{3SAT} \notin \mathbf{NP}$ is known as the "$\mathbf{NP} \neq \mathbf{co} - \mathbf{NP}$" conjecture. It implies the $\mathbf{P} \neq \mathbf{NP}$ conjecture (see Exercise 15.2).

We have previously informally equated the notion of $F \leq_p G$ with F being "no harder than G" and in particular have seen in Solved Exercise 14.1 that if $G \in \mathbf{P}$ and $F \leq_p G$, then $F \in \mathbf{P}$ as well. The following exercise shows that if $F \leq_p G$ then it is also "no harder to verify" than G. That is, regardless of whether or not it is in **P**, if G has the property that solutions to it can be efficiently verified, then so does F.

Solved Exercise 15.4 — Reductions and NP. Let $F, G : \{0,1\}^* \to \{0,1\}$. Show that if $F \leq_p G$ and $G \in \mathbf{NP}$ then $F \in \mathbf{NP}$.

■

Solution:

Suppose that G is in **NP** and in particular there exists a and $V \in \mathbf{P}$ such that for every $y \in \{0,1\}^*$, $G(y) = 1 \Leftrightarrow \exists_{w \in \{0,1\}^{|y|^a}} V(yw) = 1$. Suppose also that $F \leq_p G$ and so in particular there is a n^b-time computable function R such that $F(x) = G(R(x))$ for all $x \in \{0,1\}^*$. Define V' to be a Turing machine that on input a pair (x, w) computes $y = R(x)$ and returns 1 if and only if $|w| = |y|^a$ and $V(yw) = 1$. Then V' runs in polynomial time, and for every $x \in \{0,1\}^*$, $F(x) = 1$ iff there exists w of size $|R(x)|^a$ which is at most polynomial in $|x|$ such that $V'(x, w) = 1$, hence demonstrating that $F \in \mathbf{NP}$.

■

15.2 FROM NP TO 3SAT: THE COOK-LEVIN THEOREM

We have seen several examples of problems for which we do not know if their best algorithm is polynomial or exponential, but we can show that they are in **NP**. That is, we don't know if they are easy to *solve*, but we do know that it is easy to *verify* a given solution. There are many, many, *many*, more examples of interesting functions we would like to compute that are easily shown to be in **NP**. What is quite amazing is that if we can solve 3SAT then we can solve all of them!

The following is one of the most fundamental theorems in Computer Science:

Theorem 15.6 — Cook-Levin Theorem. For every $F \in \mathbf{NP}$, $F \leq_p 3SAT$.

We will soon show the proof of Theorem 15.6, but note that it immediately implies that *QUADEQ*, *LONGPATH*, and *MAXCUT* all reduce to $3SAT$. Combining it with the reductions we've seen in Chapter 14, it implies that all these problems are *equivalent!* For example, to reduce *QUADEQ* to *LONGPATH*, we can first reduce *QUADEQ* to $3SAT$ using Theorem 15.6 and use the reduction we've seen in Theorem 14.12 from $3SAT$ to *LONGPATH*. That is, since $QUADEQ \in \mathbf{NP}$, Theorem 15.6 implies that $QUADEQ \leq_p 3SAT$, and Theorem 14.12 implies that $3SAT \leq_p LONGPATH$, which by the transitivity of reductions (Solved Exercise 14.2) means that $QUADEQ \leq_p LONGPATH$. Similarly, since $LONGPATH \in \mathbf{NP}$, we can use Theorem 15.6 and Theorem 14.4 to show that $LONGPATH \leq_p 3SAT \leq_p QUADEQ$, concluding that *LONGPATH* and *QUADEQ* are computationally equivalent.

There is of course nothing special about *QUADEQ* and *LONGPATH* here: by combining (15.6) with the reductions we saw, we see that just like $3SAT$, *every* $F \in \mathbf{NP}$ reduces to *LONGPATH*, and the same is true for *QUADEQ* and *MAXCUT*. All these problems are in some sense "the hardest in **NP**" since an efficient algorithm for any one of them would imply an efficient algorithm for *all* the problems in **NP**. This motivates the following definition:

Definition 15.7 — NP-hardness and NP-completeness. Let $G : \{0,1\}^* \to \{0,1\}$. We say that G is **NP** *hard* if for every $F \in \mathbf{NP}$, $F \leq_p G$.

We say that G is **NP** *complete* if G is **NP** hard and $G \in \mathbf{NP}$.

The Cook-Levin Theorem (Theorem 15.6) can be rephrased as saying that $3SAT$ is **NP** hard, and since it is also in **NP**, this means that $3SAT$ is **NP** complete. Together with the reductions of Chapter 14, Theorem 15.6 shows that despite their superficial differences, 3SAT, quadratic equations, longest path, independent set, and maximum

cut, are all **NP**-complete. Many thousands of additional problems have been shown to be **NP**-complete, arising from all the sciences, mathematics, economics, engineering and many other fields. (For a few examples, see this Wikipedia page and this website.)

Big Idea 22 If a *single* **NP**-complete has a polynomial-time algorithm, then there is such an algorithm for every decision problem that corresponds to the existence of an *efficiently-verifiable* solution.

15.2.1 What does this mean?

As we've seen in Solved Exercise 15.2, $\mathbf{P} \subseteq \mathbf{NP}$. *The* most famous conjecture in Computer Science is that this containment is *strict*. That is, it is widely conjectured that $\mathbf{P} \neq \mathbf{NP}$. One way to refute the conjecture that $\mathbf{P} \neq \mathbf{NP}$ is to give a polynomial-time algorithm for even a single one of the **NP**-complete problems such as 3SAT, Max Cut, or the thousands of others that have been studied in all fields of human endeavors. The fact that these problems have been studied by so many people, and yet not a single polynomial-time algorithm for any of them has been found, supports that conjecture that indeed $\mathbf{P} \neq \mathbf{NP}$. In fact, for many of these problems (including all the ones we mentioned above), we don't even know of a $2^{o(n)}$-time algorithm! However, to the frustration of computer scientists, we have not yet been able to prove that $\mathbf{P} \neq \mathbf{NP}$ or even rule out the existence of an $O(n)$-time algorithm for 3SAT. Resolving whether or not $\mathbf{P} = \mathbf{NP}$ is known as the P vs NP problem. A million-dollar prize has been offered for the solution of this problem, a popular book has been written, and every year a new paper comes out claiming a proof of $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$, only to wither under scrutiny.

One of the mysteries of computation is that people have observed a certain empirical "zero-one law" or "dichotomy" in the computational complexity of natural problems, in the sense that many natural problems are either in **P** (often in $TIME(O(n))$ or $TIME(O(n^2))$), or they are are **NP** hard. This is related to the fact that for most natural problems, the best known algorithm is either exponential or polynomial, with not too many examples where the best running time is some strange intermediate complexity such as $2^{2^{\sqrt{\log n}}}$. However, it is believed that there exist problems in **NP** that are neither in **P** nor are **NP**-complete, and in fact a result known as "Ladner's Theorem" shows that if $\mathbf{P} \neq \mathbf{NP}$ then this is indeed the case (see also Exercise 15.1 and Fig. 15.3).

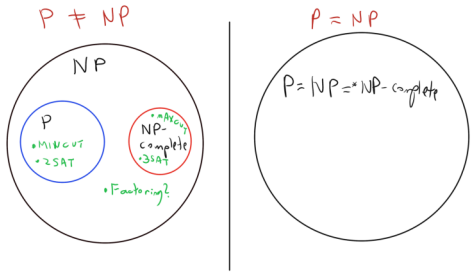


Figure 15.3: The world if $\mathbf{P} \neq \mathbf{NP}$ (left) and $\mathbf{P} = \mathbf{NP}$ (right). In the former case the set of **NP**-complete problems is disjoint from **P** and Ladner's theorem shows that there exist problems that are neither in **P** nor are **NP**-complete. (There are remarkably few natural candidates for such problems, with some prominent examples being decision variants of problems such as integer factoring, lattice shortest vector, and finding Nash equilibria.) In the latter case that $\mathbf{P} = \mathbf{NP}$ the notion of **NP**-completeness loses its meaning, as essentially all functions in **P** (save for the trivial constant zero and constant one functions) are **NP**-complete.

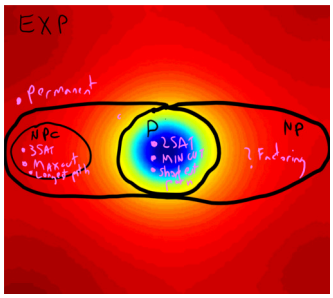


Figure 15.4: A rough illustration of the (conjectured) status of problems in exponential time. Darker colors correspond to higher running time, and the circle in the middle is the problems in **P**. **NP** is a (conjectured to be proper) superclass of **P** and the **NP**-complete problems (or **NPC** for short) are the "hardest" problems in **NP**, in the sense that a solution for one of them implies a solution for all other problems in **NP**. It is conjectured that all the **NP**-complete problems require at least $\exp(n^{\epsilon})$ time to solve for a constant $\epsilon > 0$, and many require $\exp(\Omega(n))$ time. The *permanent* is not believed to be contained in **NP** though it is **NP**-hard, which means that a polynomial-time algorithm for it implies that $\mathbf{P} = \mathbf{NP}$.

15.2.2 The Cook-Levin Theorem: Proof outline

We will now prove the Cook-Levin Theorem, which is the underpinning to a great web of reductions from 3SAT to thousands of problems across many great fields. Some problems that have been shown to be **NP**-complete include: minimum-energy protein folding, minimum surface-area foam configuration, map coloring, optimal Nash equilibrium, quantum state entanglement, minimum supersequence of a genome, minimum codeword problem, shortest vector in a lattice, minimum genus knots, positive Diophantine equations, integer programming, and many many more. The worst-case complexity of all these problems is (up to polynomial factors) equivalent to that of 3SAT, and through the Cook-Levin Theorem, to all problems in **NP**.

To prove Theorem 15.6 we need to show that $F \leq_p 3SAT$ for every $F \in$ **NP**. We will do so in three stages. We define two intermediate problems: *NANDSAT* and *3NAND*. We will shortly show the definitions of these two problems, but Theorem 15.6 will follow from combining the following three results:

1. *NANDSAT* is **NP** hard (Lemma 15.8).
2. $NANDSAT \leq_p 3NAND$ (Lemma 15.9).
3. $3NAND \leq_p 3SAT$ (Lemma 15.10).

By the transitivity of reductions, it will follow that for every $F \in$ **NP**,

$$F \leq_p NANDSAT \leq_p 3NAND \leq_p 3SAT$$

hence establishing Theorem 15.6.

We will prove these three results Lemma 15.8, Lemma 15.9 and Lemma 15.10 one by one, providing the requisite definitions as we go along.

15.3 THE *NANDSAT* PROBLEM, AND WHY IT IS NP HARD

The function $NANDSAT : \{0,1\}^* \to \{0,1\}$ is defined as follows:

- The **input** to *NANDSAT* is a string Q representing a NAND-CIRC program (or equivalently, a circuit with NAND gates).
- The **output** of *NANDSAT* on input Q is 1 if and only if there exists a string $w \in \{0,1\}^n$ (where n is the number of inputs to Q) such that $Q(w) = 1$.

Solved Exercise 15.5 — $NANDSAT \in$ NP. Prove that $NANDSAT \in$ **NP**.

■

Solution:

We have seen that the circuit (or straightline program) evaluation problem can be computed in polynomial time. Specifically, given a NAND-CIRC program Q of s lines and n inputs, and $w \in \{0,1\}^n$, we can evaluate Q on the input w in time which is polynomial in s and hence verify whether or not $Q(w) = 1$.

■

We now prove that *NANDSAT* is **NP** hard.

Lemma 15.8 *NANDSAT* is **NP** hard.

Proof Idea:

The proof closely follows the proof that $\mathbf{P} \subseteq \mathbf{P_{/poly}}$ (Theorem 13.12 , see also Section 13.6.2). Specifically, if $F \in \mathbf{NP}$ then there is a polynomial time Turing machine M and positive integer a such that for every $x \in \{0,1\}^n$, $F(x) = 1$ iff there is some $w \in \{0,1\}^{n^a}$ such that $M(xw) = 1$. The proof that $\mathbf{P} \subseteq \mathbf{P_{/poly}}$ gave us a way (via "unrolling the loop") to come up in polynomial time with a Boolean circuit C on n^a inputs that computes the function $w \mapsto M(xw)$. We can then translate C into an equivalent NAND circuit (or NAND-CIRC program) Q. We see that there is a string $w \in \{0,1\}^{n^a}$ such that $Q(w) = 1$ if and only if there is such w satisfying $M(xw) = 1$ which (by definition) happens if and only if $F(x) = 1$. Hence the translation of x into the circuit Q is a reduction showing $F \leq_p$ *NANDSAT*.

⋆

P

The proof is a little bit technical but ultimately follows quite directly from the definition of **NP**, as well as the ability to "unroll the loop" of NAND-TM programs as discussed in Section 13.6.2. If you find it confusing, try to pause here and think how you would implement in your favorite programming language the function `unroll` which on input a NAND-TM program P and numbers T, n outputs an n-input NAND-CIRC program Q of $O(|T|)$ lines such that for every input $z \in \{0,1\}^n$, if P halts on z within at most T steps and outputs y, then $Q(z) = y$.

Proof of Lemma 15.8. Let $F \in \mathbf{NP}$. To prove Lemma 15.8 we need to give a polynomial-time computable function that will map every $x^* \in \{0,1\}^*$ to a NAND-CIRC program Q such that $F(x) = NANDSAT(Q)$.

Let $x^* \in \{0,1\}^*$ be such a string and let $n = |x^*|$ be its length. By Definition 15.1 there exists $V \in \mathbf{P}$ and positive $a \in \mathbb{N}$ such that $F(x^*) = 1$ if and only if there exists $w \in \{0,1\}^{n^a}$ satisfying $V(x^*w) = 1$.

Let $m = n^a$. Since $V \in \mathbf{P}$ there is some NAND-TM program P^* that computes V on inputs of the form xw with $x \in \{0,1\}^n$ and $w \in \{0,1\}^m$ in at most $(n+m)^c$ time for some constant c. Using our "unrolling the loop NAND-TM to NAND compiler" of Theorem 13.14, we can obtain a NAND-CIRC program Q' that has $n+m$ inputs and at most $O((n+m)^{2c})$ lines such that $Q'(xw) = P^*(xw)$ for every $x \in \{0,1\}^n$ and $w \in \{0,1\}^m$.

We can then use a simple "hardwiring" technique, reminiscent of Remark 9.11 to map Q' into a circuit/NAND-CIRC program Q on m inputs such that $Q(w) = Q'(x^*w)$ for every $w \in \{0,1\}^m$.

CLAIM: There is a polynomial-time algorithm that on input a NAND-CIRC program Q' on $n+m$ inputs and $x^* \in \{0,1\}^n$, outputs a NAND-CIRC program Q such that for every $w \in \{0,1\}^n$, $Q(w) = Q'(x^*w)$.

PROOF OF CLAIM: We can do so by adding a few lines to ensure that the variables `zero` and `one` are 0 and 1 respectively, and then simply replacing any reference in Q' to an input x_i with $i \in [n]$ the corresponding value based on x^*_i. See Fig. 15.5 for an implementation of this reduction in Python.

Our final reduction maps an input x^*, into the NAND-CIRC program Q obtained above. By the above discussion, this reduction runs in polynomial time. Since we know that $F(x^*) = 1$ if and only if there exists $w \in \{0,1\}^m$ such that $P^*(x^*w) = 1$, this means that $F(x^*) = 1$ if and only if $NANDSAT(Q) = 1$, which is what we wanted to prove. ■

```python
# "hardwire" an input x to a NAND program Q
# return Q' s.t. Q'(y)=Q(xy)
def hardwire(Q,x):
    n = len(x)

    for i in range(n):
        Q = Q.replace(f'X[{i}]',('one' if x[i] else 'zero'))

    # move x_n,x_n+1,.... to x_0,x_1,...
    for i in range(n,2*len(Q)):
        Q = Q.replace(f'X[{i}]',f'X[{i-n}]')

    return CONSTPREFIX+Q

CONSTPREFIX = r'''
temp = NAND(X[0],X[0])
one = NAND(X[0],temp)
zero = NAND(one,one)
'''[1:]
```

```python
print(xor5)
```

```
u   =  NAND(X[0],X[1])
v   = NAND( X[0] , u)
w   = NAND( X[1] , u)
s   = NAND( v   , w)
u   = NAND( s   , X[2])
v   = NAND( s   , u)
w   = NAND( X[2] , u)
s   = NAND( v   , w)
u   = NAND( s   , X[3])
v   = NAND( s   , u)
w   = NAND( X[3] , u)
s   = NAND( v   , w)
u   = NAND( s   , X[4])
v   = NAND( s   , u)
w   = NAND( X[4] , u)
Y[0] = NAND( v   , w)
```

```python
print(hardwire(xor5,[1,0,1]))
```

```
temp = NAND(X[0],X[0])
one = NAND(X[0],temp)
zero = NAND(one,one)
u   =  NAND(one,zero)
v   = NAND( one , u)
w   = NAND( zero , u)
s   = NAND( v   , w)
u   = NAND( s   , one)
v   = NAND( s   , u)
w   = NAND( one , u)
s   = NAND( v   , w)
u   = NAND( s   , X[0])
v   = NAND( s   , u)
w   = NAND( X[0] , u)
s   = NAND( v   , w)
u   = NAND( s   , X[1])
v   = NAND( s   , u)
w   = NAND( X[1] , u)
Y[0] = NAND( v   , w)
```

Figure 15.5: Given an T-line NAND-CIRC program Q that has $n+m$ inputs and some $x^* \in \{0,1\}^n$, we can transform Q into a $T+3$ line NAND-CIRC program Q' that computes the map $w \mapsto Q(x^*w)$ for $w \in \{0,1\}^m$ by simply adding code to compute the `zero` and `one` constants, replacing all references to `X[`i`]` with either `zero` or `one` depending on the value of x^*_i, and then replacing the remaining references to `X[`j`]` with `X[`$j-n$`]`. Above is Python code that implements this transformation, as well as an example of its execution on a simple program.

15.4 THE 3*NAND* PROBLEM

The 3*NAND* problem is defined as follows:

- The **input** is a logical formula Ψ on a set of variables $z_0, \ldots, z_{r-1}$ which is an AND of constraints of the form $z_i = NAND(z_j, z_k)$.

- The **output** is 1 if and only if there is an input $z \in \{0,1\}^r$ that satisfies all of the constraints.

For example, the following is a $3NAND$ formula with 5 variables and 3 constraints:

$$\Psi = (z_3 = NAND(z_0, z_2)) \wedge (z_1 = NAND(z_0, z_2)) \wedge (z_4 = NAND(z_3, z_1)) \ .$$

In this case $3NAND(\Psi) = 1$ since the assignment $z = 01010$ satisfies it. Given a $3NAND$ formula Ψ on r variables and an assignment $z \in \{0,1\}^r$, we can check in polynomial time whether $\Psi(z) = 1$, and hence $3NAND \in$ **NP**. We now prove that $3NAND$ is **NP** hard:

Lemma 15.9 $NANDSAT \leq_p 3NAND$.

Proof Idea:

To prove Lemma 15.9 we need to give a polynomial-time map from every NAND-CIRC program Q to a 3NAND formula Ψ such that there exists w such that $Q(w) = 1$ if and only if there exists z satisfying Ψ. For every line i of Q, we define a corresponding variable z_i of Ψ. If the line i has the form `foo = NAND(bar,blah)` then we will add the clause $z_i = NAND(z_j, z_k)$ where j and k are the last lines in which `bar` and `blah` were written to. We will also set variables corresponding to the input variables, as well as add a clause to ensure that the final output is 1. The resulting reduction can be implemented in about a dozen lines of Python, see Fig. 15.6.

⋆

```
# Reduce NANDSAT to 3NAND
# Input: NAND prog Q
# Output: 3NAND formula Ψ
#         s.t. Ψ satisfiable iff Q is
def NANDSAT23NAND_(Q):
    Q = CONSTPREFIX + Q
    n, _ = numinout(Q)
    Ψ = ''

    #varidx[u] is n+line where u is last written to
    varidx = defaultdict(lambda : n+ 2) # line 2 corresponds to zero

    for i in range(n): varidx[f'X[{i}]'] = i # setup x_0...x_n-1

    j = n
    for line in Q.split('\n'):
        if not line.strip(): continue
        foo,bar,blah = splitline(line) # split "foo = NAND(bar,blah)"
        Ψ +=  f"(z{j} = NAND(z{varidx[bar]},z{varidx[blah]}) ) ∧ "
        varidx[foo] = j
        j += 1
    Ψ += f"(z{varidx['Y[0]']} = NAND(z{varidx['zero']},z{varidx['zero']}) )"
    return Ψ
```

```
print(xor5)
```

```
u   =  NAND(X[0],X[1])
v   = NAND( X[0] , u)
w   = NAND( X[1] , u)
s   = NAND( v    , w)
u   = NAND( s    , X[2])
v   = NAND( s    , u)
w   = NAND( X[2] , u)
s   = NAND( v    , w)
u   = NAND( s    , X[3])
v   = NAND( s    , u)
w   = NAND( X[3] , u)
s   = NAND( v    , w)
u   = NAND( s    , X[4])
v   = NAND( s    , u)
w   = NAND( X[4] , u)
Y[0] = NAND( v    , w)
```

```
print(NANDSAT23NAND_(xor5))
```

```
(z5 = NAND(z0,z0) ) ∧ (z6 = NAND(z0,z5) ) ∧ (z7 = NAND(z6,z6) ) ∧ (z8 = NAND(z0,z1) ) ∧ (z9 = NAND(z
0,z8) ) ∧ (z10 = NAND(z1,z8) ) ∧ (z11 = NAND(z9,z10) ) ∧ (z12 = NAND(z11,z2) ) ∧ (z13 = NAND(z11,z1
2) ) ∧ (z14 = NAND(z2,z12) ) ∧ (z15 = NAND(z13,z14) ) ∧ (z16 = NAND(z15,z3) ) ∧ (z17 = NAND(z15,z16)
) ∧ (z18 = NAND(z3,z16) ) ∧ (z19 = NAND(z17,z18) ) ∧ (z20 = NAND(z19,z4) ) ∧ (z21 = NAND(z19,z20) )
∧ (z22 = NAND(z4,z20) ) ∧ (z23 = NAND(z21,z22) ) ∧ (z23 = NAND(z7,z7) )
```

```
eval3NAND(NANDSAT23NAND_(xor5),[1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1])
```

```
True
```

Figure 15.6: Python code to reduce an instance Q of $NANDSAT$ to an instance Ψ of $3NAND$. In the example above we transform the NAND-CIRC program `xor5` which has 5 input variables and 16 lines, into a $3NAND$ formula Ψ that has 24 variables and 20 clauses. Since `xor5` outputs 1 on the input $1, 0, 0, 1, 1$, there exists an assignment $z \in \{0,1\}^{24}$ to Ψ such that $(z_0, z_1, z_2, z_3, z_4) = (1, 0, 0, 1, 1)$ and Ψ evaluates to *true* on z.

Proof of Lemma 15.9. To prove Lemma 15.9 we need to give a reduction from *NANDSAT* to *3NAND*. Let Q be a NAND-CIRC program with n inputs, one output, and m lines. We can assume without loss of generality that Q contains the variables `one` and `zero` as usual.

We map Q to a *3NAND* formula Ψ as follows:

- Ψ has $m+n$ variables $z_0, \ldots, z_{m+n-1}$.
- The first n variables $z_0, \ldots, z_{n-1}$ will corresponds to the inputs of Q. The next m variables $z_n, \ldots, z_{n+m-1}$ will correspond to the m lines of Q.
- For every $\ell \in \{n, n+1, \ldots, n+m\}$, if the $\ell - n$-th line of the program Q is `foo = NAND(bar,blah)` then we add to Ψ the constraint $z_\ell = NAND(z_j, z_k)$ where $j-n$ and $k-n$ correspond to the last lines in which the variables `bar` and `blah` (respectively) were written to. If one or both of `bar` and `blah` was not written to before then we use z_{ℓ_0} instead of the corresponding value z_j or z_k in the constraint, where $\ell_0 - n$ is the line in which `zero` is assigned a value. If one or both of `bar` and `blah` is an input variable `X[i]` then we use z_i in the constraint.
- Let ℓ^* be the last line in which the output `y_0` is assigned a value. Then we add the constraint $z_{\ell^*} = NAND(z_{\ell_0}, z_{\ell_0})$ where $\ell_0 - n$ is as above the last line in which `zero` is assigned a value. Note that this is effectively the constraint $z_{\ell^*} = NAND(0,0) = 1$.

To complete the proof we need to show that there exists $w \in \{0,1\}^n$ s.t. $Q(w) = 1$ if and only if there exists $z \in \{0,1\}^{n+m}$ that satisfies all constraints in Ψ. We now show both sides of this equivalence.

Part I: Completeness. Suppose that there is $w \in \{0,1\}^n$ s.t. $Q(w) = 1$. Let $z \in \{0,1\}^{n+m}$ be defined as follows: for $i \in [n]$, $z_i = w_i$ and for $i \in \{n, n+1, \ldots, n+m\}$ z_i equals the value that is assigned in the $(i-n)$-th line of Q when executed on w. Then by construction z satisfies all of the constraints of Ψ (including the constraint that $z_{\ell^*} = NAND(0,0) = 1$ since $Q(w) = 1$.)

Part II: Soundness. Suppose that there exists $z \in \{0,1\}^{n+m}$ satisfying Ψ. Soundness will follow by showing that $Q(z_0, \ldots, z_{n-1}) = 1$ (and hence in particular there exists $w \in \{0,1\}^n$, namely $w = z_0 \cdots z_{n-1}$, such that $Q(w) = 1$). To do this we will prove the following claim $(*)$: for every $\ell \in [m]$, $z_{\ell+n}$ equals the value assigned in the ℓ-th step of the execution of the program Q on $z_0, \ldots, z_{n-1}$. Note that because z satisfies the constraints of Ψ, $(*)$ is sufficient to prove the soundness condition since these constraints imply that the last value assigned to the variable `y_0` in the execution of Q on $z_0 \cdots w_{n-1}$ is equal to 1. To prove $(*)$ suppose, towards a contradiction, that it is false, and let ℓ be

the smallest number such that $z_{\ell+n}$ is *not* equal to the value assigned in the ℓ-th step of the execution of Q on $z_0, \ldots, z_{n-1}$. But since z satisfies the constraints of Ψ, we get that $z_{\ell+n} = NAND(z_i, z_j)$ where (by the assumption above that ℓ is *smallest* with this property) these values *do* correspond to the values last assigned to the variables on the right-hand side of the assignment operator in the ℓ-th line of the program. But this means that the value assigned in the ℓ-th step is indeed simply the NAND of z_i and z_j, contradicting our assumption on the choice of ℓ.

■

15.5 FROM $3NAND$ TO $3SAT$

The final step in the proof of Theorem 15.6 is the following:

Lemma 15.10 $3NAND \leq_p 3SAT$.

Proof Idea:

To prove Lemma 15.10 we need to map a 3NAND formula φ into a 3SAT formula ψ such that φ is satisfiable if and only if ψ is. The idea is that we can transform every NAND constraint of the form $a = NAND(b, c)$ into the AND of ORs involving the variables a, b, c and their negations, where each of the ORs contains at most three terms. The construction is fairly straightforward, and the details are given below.

⋆

P

It is a good exercise for you to try to find a 3CNF formula ξ on three variables a, b, c such that $\xi(a, b, c)$ is true if and only if $a = NAND(b, c)$. Once you do so, try to see why this implies a reduction from $3NAND$ to $3SAT$, and hence completes the proof of Lemma 15.10

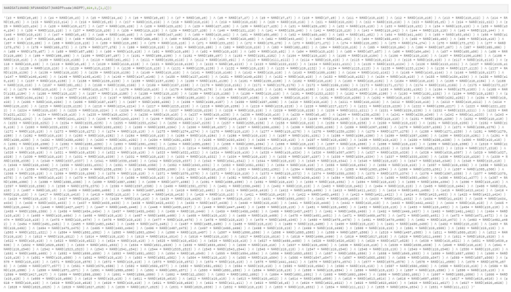


Figure 15.7: A $3NAND$ instance that is obtained by taking a NAND-TM program for computing the AND function, unrolling it to obtain a $NANDSAT$ instance, and then composing it with the reduction of Lemma 15.9.

```
# Reduce 3NAND to 3SAT
# Input: 3NAND formula Ψ
# Output: 3CNF formula φ
#         s.t.  φ satisfiable iff Ψ is
def NAND23SAT_(Ψ):
    ϕ = ""
    for (a,b,c) in getnandclauses(Ψ):
        ϕ += f'(¬{a} ∨ ¬{b} ∨ ¬{c}) ∧ ({a} ∨ {b}  ∨ {b}) ∧ ({a}  ∨  {c} ∨ {c}) ∧ '
    return ϕ[:-3] # chop off redundant ∧
```

```
Ψ = "(x0 = NAND(x2,x3) ) ∧ (x3 = NAND(x2,x1) ) ∧ (x1 = NAND(x2,x3) ) "
NAND23SAT_(Ψ)
```

```
'(¬x0 ∨ ¬x2 ∨ ¬x3) ∧ (x0 ∨ x2  ∨ x2) ∧ (x0  ∨  x3 ∨ x3) ∧ (¬x3 ∨ ¬x2 ∨ ¬x1) ∧ (x3 ∨ x2  ∨ x2) ∧ (x3
∨  x1 ∨ x1) ∧ (¬x1 ∨ ¬x2 ∨ ¬x3) ∧ (x1 ∨ x2  ∨ x2) ∧ (x1  ∨  x3 ∨ x3)'
```

Figure 15.8: Code and example output for the reduction given in Lemma 15.10 of $3NAND$ to $3SAT$.

Proof of Lemma 15.10. The constraint

$$z_i = NAND(z_j, z_k) \tag{15.2}$$

is satisfied if $z_i = 1$ whenever $(z_j, z_k) \neq (1, 1)$. By going through all cases, we can verify that (15.2) is equivalent to the constraint

$$(\overline{z_i} \vee \overline{z_j} \vee \overline{z_k}) \wedge (z_i \vee z_j) \wedge (z_i \vee z_k) \ . \tag{15.3}$$

Indeed if $z_j = z_k = 1$ then the first constraint of Eq. (15.3) is only true if $z_i = 0$. On the other hand, if either of z_j or z_k equals 0 then unless $z_i = 1$ either the second or third constraints will fail. This means that, given any 3NAND formula φ over n variables $z_0, \ldots, z_{n-1}$, we can obtain a 3SAT formula ψ over the same variables by replacing every 3*NAND* constraint of φ with three 3*OR* constraints as in Eq. (15.3).[1] Because of the equivalence of (15.2) and (15.3), the formula ψ satisfies that $\psi(z_0, \ldots, z_{n-1}) = \varphi(z_0, \ldots, z_{n-1})$ for every assignment $z_0, \ldots, z_{n-1} \in \{0,1\}^n$ to the variables. In particular ψ is satisfiable if and only if φ is, thus completing the proof. ■

[1] The resulting formula will have some of the OR's involving only two variables. If we wanted to insist on each formula involving three distinct variables we can always add a "dummy variable" z_{n+m} and include it in all the OR's involving only two variables, and add a constraint requiring this dummy variable to be zero.

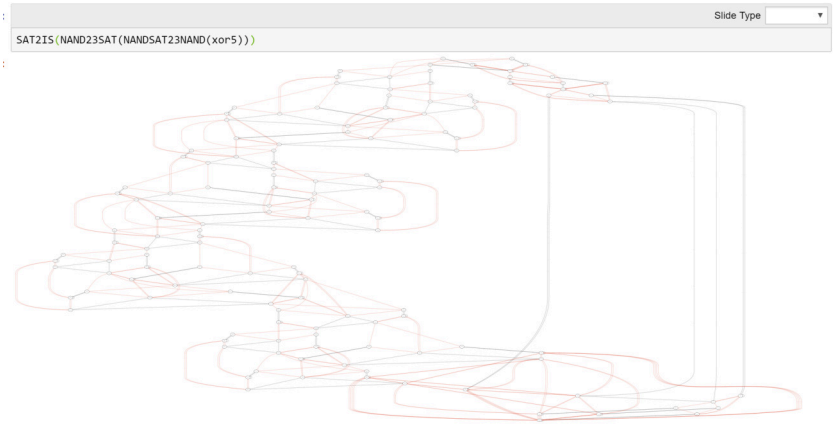


Figure 15.9: An instance of the *independent set* problem obtained by applying the reductions $NANDSAT \leq_p 3NAND \leq_p 3SAT \leq_p ISAT$ starting with the `xor5` NAND-CIRC program.

15.6 WRAPPING UP

We have shown that for every function F in **NP**, $F \leq_p NANDSAT \leq_p 3NAND \leq_p 3SAT$, and so $3SAT$ is **NP**-hard. Since in Chapter 14 we saw that $3SAT \leq_p QUADEQ$, $3SAT \leq_p ISET$, $3SAT \leq_p MAXCUT$ and $3SAT \leq_p LONGPATH$, all these problems are **NP**-hard as well. Finally, since all the aforementioned problems are in **NP**, they are all in fact **NP**-complete and have equivalent complexity. There are thousands of other natural problems that are **NP**-complete as well. Finding a polynomial-time algorithm for any one of them will imply a polynomial-time algorithm for all of them.

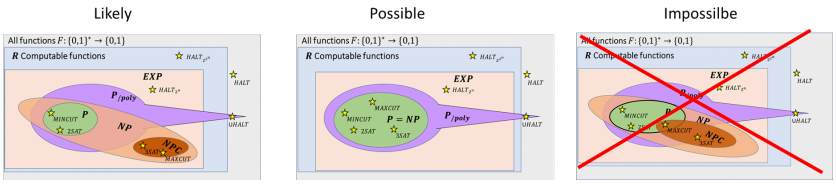


Figure 15.10: We believe that $\mathbf{P} \neq \mathbf{NP}$ and all **NP** complete problems lie outside of **P**, but we cannot rule out the possiblity that $\mathbf{P} = \mathbf{NP}$. However, we can rule out the possiblity that *some* **NP**-complete problems are in **P** and others are not, since we know that if even one **NP**-complete problem is in **P** then $\mathbf{P} = \mathbf{NP}$. The relation between $\mathbf{P}_{/\mathbf{poly}}$ and **NP** is not known though it can be shown that if one **NP**-complete problem is in $\mathbf{P}_{/\mathbf{poly}}$ then $\mathbf{NP} \subseteq \mathbf{P}_{/\mathbf{poly}}$.

Chapter Recap

- Many of the problems for which we don't know polynomial-time algorithms are **NP**-complete, which means that finding a polynomial-time algorithm for one of them would imply a polynomial-time algorithm for *all* of them.
- It is conjectured that $\mathbf{NP} \neq \mathbf{P}$ which means that we believe that polynomial-time algorithms for these problems are not merely *unknown* but are *non-existent*.
- While an **NP**-hardness result means for example that a full-fledged "textbook" solution to a problem such as MAX-CUT that is as clean and general as the algorithm for MIN-CUT probably does not exist, it does not mean that we need to give up whenever we see a MAX-CUT instance. Later in this course we will discuss several strategies to deal with **NP**-hardness, including *average-case complexity* and *approximation algorithms*.

15.7 EXERCISES

Exercise 15.1 — Poor man's Ladner's Theorem. Prove that if there is no $n^{O(\log^2 n)}$ time algorithm for $3SAT$ then there is some $F \in \mathbf{NP}$ such that $F \notin \mathbf{P}$ and F is not **NP** complete.[2]

Exercise 15.2 — $\mathbf{NP} \neq \mathbf{co}-\mathbf{NP} \Rightarrow \mathbf{NP} \neq \mathbf{P}$. Let $\overline{3SAT}$ be the function that on input a 3CNF formula φ return $1 - 3SAT(\varphi)$. Prove that if $\overline{3SAT} \notin \mathbf{NP}$ then $\mathbf{P} \neq \mathbf{NP}$. See footnote for hint.[3]

Exercise 15.3 Define $WSAT$ to be the following function: the input is a CNF formula φ where each clause is the OR of one to three variables (*without negations*), and a number $k \in \mathbb{N}$. For example, the following formula can be used for a valid input to $WSAT$: $\varphi = (x_5 \vee x_2 \vee x_1) \wedge (x_1 \vee x_3 \vee x_0) \wedge (x_2 \vee x_4 \vee x_0)$. The output $WSAT(\varphi, k) = 1$ if and only if there exists a satisfying assignment to φ in which exactly k of the variables get the value 1. For example for the formula above

[2] **Hint:** Use the function F that on input a formula φ and a string of the form 1^t, outputs 1 if and only if φ is satisfiable and $t = |\varphi|^{\log|\varphi|}$.

[3] *Hint:* Prove and then use the fact that **P** *is* closed under complement.

$WSAT(\varphi, 2) = 1$ since the assignment $(1, 1, 0, 0, 0, 0)$ satisfies all the clauses. However $WSAT(\varphi, 1) = 0$ since there is no single variable appearing in all clauses.

Prove that *WSAT* is **NP**-complete.

■

Exercise 15.4 In the *employee recruiting problem* we are given a list of potential employees, each of which has some subset of m potential skills, and a number k. We need to assemble a team of k employees such that for every skill there would be one member of the team with this skill.

For example, if Alice has the skills "C programming", "NAND programming" and "Solving Differential Equations", Bob has the skills "C programming" and "Solving Differential Equations", and Charlie has the skills "NAND programming" and "Coffee Brewing", then if we want a team of two people that covers all the four skills, we would hire Alice and Charlie.

Define the function *EMP* s.t. on input the skills L of all potential employees (in the form of a sequence L of n lists $L_1, \ldots, L_n$, each containing distinct numbers between 0 and m), and a number k, $EMP(L, k) = 1$ if and only if there is a subset S of k potential employees such that for every skill j in $[m]$, there is an employee in S that has the skill j.

Prove that *EMP* is **NP** complete.

■

Exercise 15.5 — Balanced max cut. Prove that the "balanced variant" of the maximum cut problem is **NP**-complete, where this is defined as $BMC : \{0,1\}^* \to \{0,1\}$ where for every graph $G = (V, E)$ and $k \in \mathbb{N}$, $BMC(G, k) = 1$ if and only if there exists a cut S in G cutting at least k edges such that $|S| = |V|/2$.

■

Exercise 15.6 — Regular expression intersection. Let *MANYREGS* be the following function: On input a list of regular expressions $exp_0, \ldots, \exp_m$ (represented as strings in some standard way), output 1 if and only if there is a single string $x \in \{0,1\}^*$ that matches all of them. Prove that *MANYREGS* is **NP**-hard.

■

15.8 BIBLIOGRAPHICAL NOTES

Aaronson's 120 page survey [Aar16] is a beautiful and extensive exposition to the **P** vs **NP** problem, its importance and status. See also as well as Chapter 3 in Wigderson's excellent book [Wig19]. Johnson [Joh12] gives a survey of the historical development of the theory of **NP** completeness. The following web page keeps a catalog of failed

attempts at settling **P** vs **NP**. At the time of this writing, it lists about 110 papers claiming to resolve the question, of which about 60 claim to prove that $\mathbf{P} = \mathbf{NP}$ and about 50 claim to prove that $\mathbf{P} \neq \mathbf{NP}$.

Ladner's Theorem was proved by Richard Ladner in 1975. Ladner, who was born to deaf parents, later switched his research focus into computing for assistive technologies, where he have made many contributions. In 2014, he wrote a personal essay on his path from theoretical CS to accessibility research.

Eugene Lawler's quote on the "mystical power of twoness" was taken from the wonderful book "The Nature of Computation" by Moore and Mertens. See also this memorial essay on Lawler by Lenstra.

Learning Objectives:

- Explore the consequences of P = NP
- *Search-to-decision* reduction: transform algorithms that solve decision version to search version for NP-complete problems.
- Optimization and learning problems
- Quantifier elimination and solving problems in the polynomial hierarchy.
- What is the evidence for P = NP vs P ≠ NP?

16
What if P equals NP?

"You don't have to believe in God, but you should believe in The Book.", Paul Erdős, 1985.[1]

"No more half measures, Walter", Mike Ehrmantraut in "Breaking Bad", 2010.

"The evidence in favor of [**P** $\neq$ **NP**] *and* [*its algebraic counterpart*] *is so overwhelming, and the consequences of their failure are so grotesque, that their status may perhaps be compared to that of physical laws rather than that of ordinary mathematical conjectures."*, Volker Strassen, laudation for Leslie Valiant, 1986.

"Suppose aliens invade the earth and threaten to obliterate it in a year's time unless human beings can find the [*fifth Ramsey number*]*. We could marshal the world's best minds and fastest computers, and within a year we could probably calculate the value. If the aliens demanded the* [*sixth Ramsey number*]*, however, we would have no choice but to launch a preemptive attack."*, Paul Erdős, as quoted by Graham and Spencer, 1990.[2]

We have mentioned that the question of whether **P** $=$ **NP**, which is equivalent to whether there is a polynomial-time algorithm for 3*SAT*, is the great open question of Computer Science. But why is it so important? In this chapter, we will try to figure out the implications of such an algorithm.

First, let us get one qualm out of the way. Sometimes people say, *"What if* **P** $=$ **NP** *but the best algorithm for 3SAT takes* n^{1000} *time?"* Well, n^{1000} is much larger than, say, $2^{0.001\sqrt{n}}$ for any input smaller than 2^{50}, as large as a harddrive as you will encounter, and so another way to phrase this question is to say "what if the complexity of 3SAT is exponential for all inputs that we will ever encounter, but then grows much smaller than that?" To me this sounds like the computer science equivalent of asking, "what if the laws of physics change completely once they are out of the range of our telescopes?". Sure, this is a valid possibility, but wondering about it does not sound like the most productive use of our time.

[1] Paul Erdős (1913-1996) was one of the most prolific mathematicians of all times. Though he was an atheist, Erdős often referred to "The Book" in which God keeps the most elegant proof of each mathematical theorem.

[2] The k-th Ramsey number, denoted as $R(k,k)$, is the smallest number n such that for every graph G on n vertices, both G and its complement contain a k-sized independent set. If **P** $=$ **NP** then we can compute $R(k,k)$ in time polynomial in 2^k, while otherwise it can potentially take closer to $2^{2^{2k}}$ steps.

So, as the saying goes, we'll keep an open mind, but not so open that our brains fall out, and assume from now on that:

- There is a mathematical god,

  and

- She does not "beat around the bush" or take "half measures".

What we mean by this is that we will consider two extreme scenarios:

- **3SAT is very easy:** $3SAT$ has an $O(n)$ or $O(n^2)$ time algorithm with a not too huge constant (say smaller than 10^6.)
- **3SAT is very hard:** $3SAT$ is exponentially hard and cannot be solved faster than $2^{\epsilon n}$ for some not too tiny $\epsilon > 0$ (say at least 10^{-6}). We can even make the stronger assumption that for every sufficiently large n, the restriction of $3SAT$ to inputs of length n cannot be computed by a circuit of fewer than $2^{\epsilon n}$ gates.

At the time of writing, the fastest known algorithm for $3SAT$ requires more than $2^{0.35n}$ to solve n variable formulas, while we do not even know how to rule out the possibility that we can compute $3SAT$ using $10n$ gates. To put it in perspective, for the case $n = 1000$ our lower and upper bounds for the computational costs are apart by a factor of about 10^{100}. As far as we know, it could be the case that 1000-variable $3SAT$ can be solved in a millisecond on a first-generation iPhone, and it can also be the case that such instances require more than the age of the universe to solve on the world's fastest supercomputer.

So far, most of our evidence points to the latter possibility of 3SAT being exponentially hard, but we have not ruled out the former possibility either. In this chapter we will explore some of the consequences of the "$3SAT$ easy" scenario.

This chapter: A non-mathy overview

This chapter shows some of the truly breathtaking consequences that would be derived from an efficient algorithm for **NP**-complete problems. We will see that such an algorithm would imply efficient algorithms for tasks including solving search problems, eliminating quantifiers, fitting data with complex models, sampling and counting, and more. While the evidence strongly suggests that such an algorithm does not exist, the tools developed in these results have nonetheless found many other applications.

16.1 SEARCH-TO-DECISION REDUCTION

A priori, having a fast algorithm for 3SAT might not seem so impressive. Sure, such an algorithm allows us to decide the satisfiability of not just 3CNF formulas but also of quadratic equations, as well as find out whether there is a long path in a graph, and solve many other decision problems. But this is not typically what we want to do. It's not enough to know *if* a formula is satisfiable: we want to discover the actual satisfying assignment. Similarly, it's not enough to find out if a graph has a long path: we want to actually *find* the path.

It turns out that if we can solve these decision problems, we can solve the corresponding search problems as well:

Theorem 16.1 — Search vs Decision. Suppose that $\mathbf{P} = \mathbf{NP}$. Then for every polynomial-time algorithm V and $a, b \in \mathbb{N}$, there is a polynomial-time algorithm $FIND_V$ such that for every $x \in \{0,1\}^n$, if there exists $y \in \{0,1\}^{an^b}$ satisfying $V(xy) = 1$, then $FIND_V(x)$ finds some string y' satisfying this condition.

To understand what the statement of Theorem 16.1 means, let us look at the special case of the *MAXCUT* problem. It is not hard to see that there is a polynomial-time algorithm *VERIFYCUT* such that $VERIFYCUT(G,k,S) = 1$ if and only if S is a subset of G's vertices that cuts at least k edges. Theorem 16.1 implies that if $\mathbf{P} = \mathbf{NP}$ then there is a polynomial-time algorithm *FINDCUT* that on input G, k outputs a set S such that $VERIFYCUT(G,k,S) = 1$ if such a set exists. This means that if $\mathbf{P} = \mathbf{NP}$, by trying all values of k we can find in polynomial time a maximum cut in any given graph. We can use a similar argument to show that if $\mathbf{P} = \mathbf{NP}$ then we can find a satisfying assignment for every satisfiable 3CNF formula, find the longest path in a graph, solve integer programming, and so and so forth.

Proof Idea:

The idea behind the proof of Theorem 16.1 is simple; let us demonstrate it for the special case of $3SAT$. (In fact, this case is not so "special"— since $3SAT$ is **NP**-complete, we can reduce the task of solving the search problem for *MAXCUT* or any other problem in **NP** to the task of solving it for $3SAT$.) Suppose that $\mathbf{P} = \mathbf{NP}$ and we are given a satisfiable 3CNF formula φ, and we now want to find a satisfying assignment y for φ. Define $3SAT_0(\varphi)$ to output 1 if there is a satisfying assignment y for φ such that its first bit is 0, and similarly define $3SAT_1(\varphi) = 1$ if there is a satisfying assignment y with $y_0 = 1$.

The key observation is that both $3SAT_0$ and $3SAT_1$ are in **NP**, and so if **P** $=$ **NP** then we can compute them in polynomial time as well. Thus we can use this to find the first bit of the satisfying assignment. We can continue in this way to recover all the bits.

⋆

Proof of Theorem 16.1. Let V be some polynomial time algorithm and $a, b \in \mathbb{N}$ some constants. Define the function $STARTSWITH_V$ as follows: For every $x \in \{0,1\}^*$ and $z \in \{0,1\}^*$, $STARTSWITH_V(x,z) = 1$ if and only if there exists some $y \in \{0,1\}^{an^b - |z|}$ (where $n = |x|$) such that $V(xzy) = 1$. That is, $STARTSWITH_V(x,z)$ outputs 1 if there is some string w of length $a|x|^b$ such that $V(x,w) = 1$ and the first $|z|$ bits of w are $z_0, \ldots, z_{\ell-1}$. Since, given x, y, z as above, we can check in polynomial time if $V(xzy) = 1$, the function $STARTSWITH_V$ is in **NP** and hence if **P** $=$ **NP** we can compute it in polynomial time.

Now for every such polynomial-time V and $a, b \in \mathbb{N}$, we can implement $FIND_V(x)$ as follows:

Algorithm 16.2 — $FIND_V$: Search to decision reduction.

Input: $x \in \{0,1\}^n$
Output: $z \in \{0,1\}^{an^b}$ s.t. $V(xz) = 1$, if such z exists. Otherwise output the empty string.

```
Initially z_0 = z_1 = ··· = z_{an^b−1} = 0.
for ℓ = 0, ..., an^b − 1 do
    Let b_0 ← STARTSWITH_V(xz_0 ··· z_{ℓ−1}0).
    Let b_1 ← STARTSWITH_V(xz_0 ··· z_{ℓ−1}1).
    if b_0 = b_1 = 0 then
        return ""
                # Can't extend xz_0 ... z_{ℓ−1} to input V accepts
    end if
    if b_0 = 1 then
        z_ℓ ← 0
            # Can extend xz_0 ... x_{ℓ−1} with 0 to accepting input
    else
        z_ℓ ← 1
            # Can extend xz_0 ... x_{ℓ−1} with 1 to accepting input
    end if
end for
return z_0, ..., z_{an^b−1}
```

To analyze Algorithm 16.2, note that it makes $2an^b$ invocations to $STARTSWITH_V$ and hence if the latter is polynomial-time, then so is Algorithm 16.2. Now suppose that x is such that there exists *some* y satisfying $V(xy) = 1$. We claim that at every step $\ell = 0, \ldots, an^b - 1$, we

maintain the invariant that there exists $y \in \{0,1\}^{an^b}$ whose first ℓ bits are z s.t. $V(xy) = 1$. Note that this claim implies the theorem, since in particular it means that for $\ell = an^b - 1$, z satisfies $V(xz) = 1$.

We prove the claim by induction. For $\ell = 0$, this holds vacuously. Now for every $\ell > 0$, if the call $STARTSWITH_V(xz_0 \cdots z_{\ell-1}0)$ returns 1, then we are guaranteed the invariant by definition of $STARTSWITH_V$. Now under our inductive hypothesis, there is $y_\ell, \ldots, y_{an^b-1}$ such that $P(xz_0, \ldots, z_{\ell-1}y_\ell, \ldots, y_{an^b-1}) = 1$. If the call to $STARTSWITH_V(xz_0 \cdots z_{\ell-1}0)$ returns 0 then it must be the case that $y_\ell = 1$, and hence when we set $z_\ell = 1$ we maintain the invariant.

■

16.2 OPTIMIZATION

Theorem 16.1 allows us to find solutions for **NP** problems if **P** = **NP**, but it is not immediately clear that we can find the *optimal* solution. For example, suppose that **P** = **NP**, and you are given a graph G. Can you find the *longest* simple path in G in polynomial time?

P

This is actually an excellent question for you to attempt on your own. That is, assuming **P** = **NP**, give a polynomial-time algorithm that on input a graph G, outputs a maximally long simple path in the graph G.

The answer is *Yes*. The idea is simple: if **P** = **NP** then we can find out in polynomial time if an n-vertex graph G contains a simple path of length n, and moreover, by Theorem 16.1, if G does contain such a path, then we can find it. (Can you see why?) If G does not contain a simple path of length n, then we will check if it contains a simple path of length $n-1$, and continue in this way to find the largest k such that G contains a simple path of length k.

The above reasoning was not specifically tailored to finding paths in graphs. In fact, it can be vastly generalized to proving the following result:

Theorem 16.3 — Optimization from P = NP. Suppose that **P** = **NP**. Then for every polynomial-time computable function $f : \{0,1\}^* \to \mathbb{N}$ (identifying $f(x)$ with natural numbers via the binary representation) there is a polynomial-time algorithm OPT such that on input $x \in \{0,1\}^*$,

$$OPT(x, 1^m) = \max_{y \in \{0,1\}^m} f(x,y) \ .$$

> Moreover under the same assumption, there is a polynomial-time algorithm *FINDOPT* such that for every $x \in \{0,1\}^*$, $FINDOPT(x, 1^m)$ outputs $y^* \in \{0,1\}^*$ such that $f(x,y^*) = \max_{y \in \{0,1\}^m} f(x,y)$.

P

The statement of Theorem 16.3 is a bit cumbersome. To understand it, think how it would subsume the example above of a polynomial time algorithm for finding the maximum length path in a graph. In this case the function f would be the map that on input a pair x, y outputs 0 if the pair (x, y) does not represent some graph and a simple path inside the graph respectively; otherwise $f(x, y)$ would equal the length of the path y in the graph x. Since a path in an n vertex graph can be represented by at most $n \log n$ bits, for every x representing a graph of n vertices, finding $\max_{y \in \{0,1\}^{n \log n}} f(x, y)$ corresponds to finding the length of the maximum simple path in the graph corresponding to x, and finding the string y^* that achieves this maximum corresponds to actually finding the path.

Proof Idea:

The proof follows by generalizing our ideas from the longest path example above. Let f be as in the theorem statement. If **P** $=$ **NP** then for every for every string $x \in \{0,1\}^*$ and number k, we can test in $poly(|x|, m)$ time whether there exists y such that $f(x, y) \geq k$, or in other words test whether $\max_{y \in \{0,1\}^m} f(x, y) \geq k$. If $f(x, y)$ is an integer between 0 and $poly(|x| + |y|)$ (as is the case in the example of longest path) then we can just try out all possibilities for k to find the maximum number k for which $\max_y f(x, y) \geq k$. Otherwise, we can use *binary search* to hone down on the right value. Once we do so, we can use search-to-decision to actually find the string y^* that achieves the maximum.

⋆

Proof of Theorem 16.3. For every f as in the theorem statement, we can define the Boolean function $F : \{0,1\}^* \to \{0,1\}$ as follows.

$$F(x, 1^m, k) = \begin{cases} 1 & \exists_{y \in \{0,1\}^m} f(x,y) \geq k \\ 0 & \text{otherwise} \end{cases}$$

Since f is computable in polynomial time, F is in **NP**, and so under our assumption that **P** $=$ **NP**, F itself can be computed in polynomial time. Now, for every x and m, we can compute the largest k such that $F(x, 1^m, k) = 1$ by a binary search. Specifically, we will do this as follows:

1. We maintain two numbers a, b such that we are guaranteed that $a \leq \max_{y \in \{0,1\}^m} f(x,y) < b$.

2. Initially we set $a = 0$ and $b = 2^{T(n)}$ where $T(n)$ is the running time of f. (A function with $T(n)$ running time can't output more than $T(n)$ bits and so can't output a number larger than $2^{T(n)}$.)

3. At each point in time, we compute the midpoint $c = \lfloor (a+b)/2 \rfloor$) and let $y = F(1^n, c)$.

   a. If $y = 1$ then we set $a = c$ and leave b as it is.

   b. If $y = 0$ then we set $b = c$ and leave a as it is.

4. We then go back to step 3, until $b \leq a + 1$.

Since $|b - a|$ shrinks by a factor of 2, within $\log_2 2^{T(n)} = T(n)$ steps, we will get to the point at which $b \leq a + 1$, and then we can simply output a. Once we find the maximum value of k such that $F(x, 1^m, k) = 1$, we can use the search to decision reduction of Theorem 16.1 to obtain the actual value $y^* \in \{0,1\}^m$ such that $f(x, y^*) = k$. ■

■ **Example 16.4 — Integer programming.** One application for Theorem 16.3 is in solving *optimization problems*. For example, the task of *linear programming* is to find $y \in \mathbb{R}^n$ that maximizes some linear objective $\sum_{i=0}^{n-1} c_i y_i$ subject to the constraint that y satisfies linear inequalities of the form $\sum_{i=0}^{n-1} a_i y_i \leq c$. As we discussed in Section 12.1.3, there is a known polynomial-time algorithm for linear programming. However, if we want to place additional constraints on y, such as requiring the coordinates of y to be *integer* or $0/1$ *valued* then the best-known algorithms run in exponential time in the worst case. However, if **P** $=$ **NP** then Theorem 16.3 tells us that we would be able to solve all problems of this form in polynomial time. For every string x that describes a set of constraints and objective, we will define a function f such that if y satisfies the constraints of x then $f(x,y)$ is the value of the objective, and otherwise we set $f(x,y) = -M$ where M is some large number. We can then use Theorem 16.3 to compute the y that maximizes $f(x,y)$ and that will give us the assignment for the variables that satisfies our constraints and maximizes the objective. (If the computation results in y such that $f(x,y) = -M$ then we can double M and try again; if the true maximum objective is achieved by some string y^*, then eventually M will be large enough so that $-M$ would be

smaller than the objective achieved by y^*, and hence when we run procedure of Theorem 16.3 we would get a value larger than $-M$.)

Remark 16.5 — Need for binary search. In many examples, such as the case of finding the longest path, we don't need to use the binary search step in Theorem 16.3, and can simply enumerate over all possible values for k until we find the correct one. One example where we do need to use this binary search step is in the case of the problem of finding a maximum length path in a *weighted* graph. This is the problem where G is a weighted graph, and every edge of G is given a weight which is a number between 0 and 2^k. Theorem 16.3 shows that we can find the maximum-weight simple path in G (i.e., simple path maximizing the sum of the weights of its edges) in time polynomial in the number of vertices and in k.

Beyond just this example there is a vast field of mathematical optimization that studies problems of the same form as in Theorem 16.3. In the context of optimization, x typically denotes a set of constraints over some variables (that can be Boolean, integer, or real valued), y encodes an assignment to these variables, and $f(x,y)$ is the value of some *objective function* that we want to maximize. Given that we don't know efficient algorithms for **NP** complete problems, researchers in optimization research study special cases of functions f (such as linear programming and semidefinite programming) where it *is* possible to optimize the value efficiently. Optimization is widely used in a great many scientific areas including: machine learning, engineering, economics and operations research.

16.2.1 Example: Supervised learning

One classical optimization task is *supervised learning*. In supervised learning we are given a list of *examples* $x_0, x_1, \ldots, x_{m-1}$ (where we can think of each x_i as a string in $\{0,1\}^n$ for some n) and the *labels* for them $y_0, \ldots, y_{m-1}$ (which we will think of simply bits, i.e., $y_i \in \{0,1\}$). For example, we can think of the x_i's as images of either dogs or cats, for which $y_i = 1$ in the former case and $y_i = 0$ in the latter case. Our goal is to come up with a *hypothesis* or *predictor* $h : \{0,1\}^n \to \{0,1\}$ such that if we are given a new example x that has an (unknown to us) label y, then with high probability h will *predict* the label. That is, with high probability it will hold that $h(x) = y$. The idea in supervised learning is to use the *Occam's Razor principle*: the simplest hypothesis that explains the data is likely

to be correct. There are several ways to model this, but one popular approach is to pick some fairly simple function $H : \{0,1\}^{k+n} \to \{0,1\}$. We think of the first k inputs as the *parameters* and the last n inputs as the example data. (For example, we can think of the first k inputs of H as specifying the weights and connections for some neural network that will then be applied on the latter n inputs.) We can then phrase the supervised learning problem as finding, given a set of labeled examples $S = \{(x_0, y_0), \ldots, (x_{m-1}, y_{m-1})\}$, the set of parameters $\theta_0, \ldots, \theta_{k-1} \in \{0,1\}$ that minimizes the number of errors made by the predictor $x \mapsto H(\theta, x)$.[3]

[3] This is often known as Empirical Risk Minimization.

In other words, we can define for every set S as above the function $F_S : \{0,1\}^k \to [m]$ such that $F_S(\theta) = \sum_{(x,y)\in S} |H(\theta, x) - y|$. Now, finding the value θ that minimizes $F_S(\theta)$ is equivalent to solving the supervised learning problem with respect to H. For every polynomial-time computable $H : \{0,1\}^{k+n} \to \{0,1\}$, the task of minimizing $F_S(\theta)$ can be "massaged" to fit the form of Theorem 16.3 and hence if **P** $=$ **NP**, then we can solve the supervised learning problem in great generality. In fact, this observation extends to essentially any learning model, and allows for finding the optimal predictors given the minimum number of examples. (This is in contrast to many current learning algorithms, which often rely on having access to an extremely large number of examples— far beyond the minimum needed, and in particular far beyond the number of examples humans use for the same tasks.)

16.2.2 Example: Breaking cryptosystems

We will discuss *cryptography* later in this course, but it turns out that if **P** $=$ **NP** then almost every cryptosystem can be efficiently broken. One approach is to treat finding an encryption key as an instance of a supervised learning problem. If there is an encryption scheme that maps a "plaintext" message p and a key θ to a "ciphertext" c, then given examples of ciphertext/plaintext pairs of the form $(c_0, p_0), \ldots, (c_{m-1}, p_{m-1})$, our goal is to find the key θ such that $E(\theta, p_i) = c_i$ where E is the encryption algorithm. While you might think getting such "labeled examples" is unrealistic, it turns out (as many amateur home-brew crypto designers learn the hard way) that this is actually quite common in real-life scenarios, and that it is also possible to relax the assumption to having more minimal prior information about the plaintext (e.g., that it is English text). We defer a more formal treatment to Chapter 21.

16.3 FINDING MATHEMATICAL PROOFS

In the context of Gödel's Theorem, we discussed the notion of a *proof system* (see Section 11.1). Generally speaking, a *proof system* can be

thought of as an algorithm $V : \{0,1\}^* \to \{0,1\}$ (known as the *verifier*) such that given a *statement* $x \in \{0,1\}^*$ and a *candidate proof* $w \in \{0,1\}^*$, $V(x,w) = 1$ if and only if w encodes a valid proof for the statement x. Any type of proof system that is used in mathematics for geometry, number theory, analysis, etc., is an instance of this form. In fact, standard mathematical proof systems have an even simpler form where the proof w encodes a *sequence* of lines $w^0, \ldots, w^m$ (each of which is itself a binary string) such that each line w^i is either an *axiom* or follows from some prior lines through an application of some *inference rule*. For example, Peano's axioms encode a set of axioms and rules for the natural numbers, and one can use them to formalize proofs in number theory. Also, there are some even stronger axiomatic systems, the most popular one being Zermelo–Fraenkel with the Axiom of Choice or ZFC for short. Thus, although mathematicians typically write their papers in natural language, proofs of number theorists can typically be translated to ZFC or similar systems, and so in particular the existence of an n-page proof for a statement x implies that there exists a string w of length $poly(n)$ (in fact often $O(n)$ or $O(n^2)$) that encodes the proof in such a system. Moreover, because verifying a proof simply involves going over each line and checking that it does indeed follow from the prior lines, it is fairly easy to do that in $O(|w|)$ or $O(|w|^2)$ (where as usual $|w|$ denotes the length of the proof w). This means that for every reasonable proof system V, the following function $\mathit{SHORTPROOF}_V : \{0,1\}^* \to \{0,1\}$ is in **NP**, where for every input of the form $x1^m$, $\mathit{SHORTPROOF}_V(x,1^m) = 1$ if and only if there exists $w \in \{0,1\}^*$ with $|w| \leq m$ s.t. $V(xw) = 1$. That is, $\mathit{SHORTPROOF}_V(x,1^m) = 1$ if there is a proof (in the system V) of length at most m bits that x is true. Thus, if **P** $=$ **NP**, then despite Gödel's Incompleteness Theorems, we can still automate mathematics in the sense of finding proofs that are not too long for every statement that has one. (Frankly speaking, if the shortest proof for some statement requires a terabyte, then human mathematicians won't ever find this proof either.) For this reason, Gödel himself felt that the question of whether $\mathit{SHORTPROOF}_V$ has a polynomial time algorithm is of great interest. As Gödel wrote in a letter to John von Neumann in 1956 (before the concept of **NP** or even "polynomial time" was formally defined):

> One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max_F \psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi \geq k \cdot n$ [for some constant $k > 0$]. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance.

> Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem,[4] the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem.

[4] The undecidability of Entscheidungsproblem refers to the uncomputability of the function that maps a statement in first order logic to 1 if and only if that statement has a proof.

For many reasonable proof systems (including the one that Gödel referred to), $SHORTPROOF_V$ is in fact **NP**-complete, and so Gödel can be thought of as the first person to formulate the **P** vs **NP** question. Unfortunately, the letter was only discovered in 1988.

16.4 QUANTIFIER ELIMINATION (ADVANCED)

If **P** $=$ **NP** then we can solve all **NP** *search* and *optimization* problems in polynomial time. But can we do more? It turns out that the answer is that *Yes we can!*

An **NP** decision problem can be thought of as the task of deciding, given some string $x \in \{0,1\}^*$ the truth of a statement of the form

$$\exists_{y \in \{0,1\}^{p(|x|)}} V(xy) = 1$$

for some polynomial-time algorithm V and polynomial $p : \mathbb{N} \to \mathbb{N}$. That is, we are trying to determine, given some string x, whether *there exists* a string y such that x and y satisfy some polynomial-time checkable condition V. For example, in the *independent set* problem, the string x represents a graph G and a number k, the string y represents some subset S of G's vertices, and the condition that we check is whether $|S| \geq k$ and there is no edge $\{u, v\}$ in G such that both $u \in S$ and $v \in S$.

We can consider more general statements such as checking, given a string $x \in \{0,1\}^*$, the truth of a statement of the form

$$\exists_{y \in \{0,1\}^{p_0(|x|)}} \forall_{z \in \{0,1\}^{p_1(|x|)}} V(xyz) = 1 \ , \tag{16.1}$$

which in words corresponds to checking, given some string x, whether *there exists* a string y such that *for every* string z, the triple (x, y, z) satisfy some polynomial-time checkable condition. We can also consider more levels of quantifiers such as checking the truth of the statement

$$\exists_{y \in \{0,1\}^{p_0(|x|)}} \forall_{z \in \{0,1\}^{p_1(|x|)}} \exists_{w \in \{0,1\}^{p_2(|x|)}} V(xyzw) = 1 \tag{16.2}$$

and so on and so forth.

For example, given an n-input NAND-CIRC program P, we might want to find the *smallest* NAND-CIRC program P' that computes the same function as P. The question of whether there is such a P' that can be described by a string of at most s bits can be phrased as

$$\exists_{P' \in \{0,1\}^s} \forall_{x \in \{0,1\}^n} P(x) = P'(x) \tag{16.3}$$

which has the form (16.1).[5] Another example of a statement involving a levels of quantifiers would be to check, given a chess position x, whether there is a strategy that guarantees that White wins within a steps. For example is $a = 3$ we would want to check if given the board position x, *there exists* a move y for White such that *for every* move z for Black *there exists* a move w for White that ends in a a checkmate.

It turns out that if **P** $=$ **NP** then we can solve these kinds of problems as well:

Theorem 16.6 — Polynomial hierarchy collapse. If **P** $=$ **NP** then for every $a \in \mathbb{N}$, polynomial $p : \mathbb{N} \to \mathbb{N}$ and polynomial-time algorithm V, there is a polynomial-time algorithm $SOLVE_{V,a}$ that on input $x \in \{0,1\}^n$ returns 1 if and only if

$$\exists_{y_0 \in \{0,1\}^m} \forall_{y_1 \in \{0,1\}^m} \cdots Q_{y_{a-1} \in \{0,1\}^m} V(xy_0y_1 \cdots y_{a-1}) = 1 \qquad (16.4)$$

where $m = p(n)$ and Q is either $\exists$ or $\forall$ depending on whether a is odd or even, respectively. [6]

[6] For the ease of notation, we assume that all the strings we quantify over have the same length $m = p(n)$, but using simple padding one can show that this captures the general case of strings of different polynomial lengths.

Proof Idea:

To understand the idea behind the proof, consider the special case where we want to decide, given $x \in \{0,1\}^n$, whether for every $y \in \{0,1\}^n$ there exists $z \in \{0,1\}^n$ such that $V(xyz) = 1$. Consider the function F such that $F(xy) = 1$ if there exists $z \in \{0,1\}^n$ such that $V(xyz) = 1$. Since V runs in polynomial-time $F \in$ **NP** and hence if **P** $=$ **NP**, then there is an algorithm V' that on input x, y outputs 1 if and only if there exists $z \in \{0,1\}^n$ such that $V(xyz) = 1$. Now we can see that the original statement we consider is true if and only if for every $y \in \{0,1\}^n$, $V'(xy) = 1$, which means it is false if and only if the following condition $(*)$ holds: there exists some $y \in \{0,1\}^n$ such that $V'(xy) = 0$. But for every $x \in \{0,1\}^n$, the question of whether the condition $(*)$ is itself in **NP** (as we assumed V' can be computed in polynomial time) and hence under the assumption that **P** $=$ **NP** we can determine in polynomial time whether the condition $(*)$, and hence our original statement, is true.

⋆

Proof of Theorem 16.6. We prove the theorem by induction. We assume that there is a polynomial-time algorithm $SOLVE_{V,a-1}$ that can solve the problem (16.4) for $a - 1$ and use that to solve the problem for a. For $a = 1$, $SOLVE_{V,a-1}(x) = 1$ iff $V(x) = 1$ which is a polynomial-time computation since V runs in polynomial time. For every x, y_0, define the statement φ_{x,y_0} to be the following:

$$\varphi_{x,y_0} = \forall_{y_1\in\{0,1\}^m}\exists_{y_2\in\{0,1\}^m}\cdots Q_{y_{a-1}\in\{0,1\}^m}V(xy_0y_1\cdots y_{a-1}) = 1$$

By the definition of $SOLVE_{V,a}$, for every $x \in \{0,1\}^n$, our goal is that $SOLVE_{V,a}(x) = 1$ if and only if there exists $y_0 \in \{0,1\}^m$ such that φ_{x,y_0} is true.

The *negation* of φ_{x,y_0} is the statement

$$\overline{\varphi}_{x,y_0} = \exists_{y_1\in\{0,1\}^m}\forall_{y_2\in\{0,1\}^m}\cdots \overline{Q}_{y_{a-1}\in\{0,1\}^m}V(xy_0y_1\cdots y_{a-1}) = 0$$

where $\overline{Q}$ is $\exists$ if Q was $\forall$ and $\overline{Q}$ is $\forall$ otherwise. (Please stop and verify that you understand why this is true, this is a generalization of the fact that if Ψ is some logical condition then the negation of $\exists_y\forall_z\Psi(y,z)$ is $\forall_y\exists_z\neg\Psi(y,z)$.)

The crucial observation is that $\overline{\varphi}_{x,y_0}$ is exactly a statement of the form we consider with $a-1$ quantifiers instead of a, and hence by our inductive hypothesis there is some polynomial time algorithm $\overline{S}$ that on input xy_0 outputs 1 if and only if $\overline{\varphi}_{x,y_0}$ is true. If we let S be the algorithm that on input x, y_0 outputs $1-\overline{S}(xy_0)$ then we see that S outputs 1 if and only if φ_{x,y_0} is true. Hence we can rephrase the original statement (16.4) as follows:

$$\exists_{y_0\in\{0,1\}^m} S(xy_0) = 1 \tag{16.5}$$

but since S is a polynomial-time algorithm, Eq. (16.5) is clearly a statement in **NP** and hence under our assumption that $\mathbf{P} = \mathbf{NP}$ there is a polynomial time algorithm that on input $x \in \{0,1\}^n$, will determine if (16.5) is true and so also if the original statement (16.4) is true.

■

The algorithm of Theorem 16.6 can also solve the search problem as well: find the value y_0 that certifies the truth of (16.4). We note that while this algorithm is in polynomial time, the exponent of this polynomial blows up quite fast. If the original NANDSAT algorithm required $\Omega(n^2)$ time, solving a levels of quantifiers would require time $\Omega(n^{2^a})$.[7]

[7] We do not know whether such loss is inherent. As far as we can tell, it's possible that the *quantified boolean formula* problem has a linear-time algorithm. We will, however, see later in this course that it satisfies a notion known as **PSPACE**-hardness that is even stronger than **NP**-hardness.

16.4.1 Application: self improving algorithm for $3SAT$

Suppose that we found a polynomial-time algorithm A for $3SAT$ that is "good but not great". For example, maybe our algorithm runs in time cn^2 for some not too small constant c. However, it's possible that the *best possible* SAT algorithm is actually much more efficient than that. Perhaps, as we guessed before, there is a circuit C^* of at most $10^6 n$ gates that computes 3SAT on n variables, and we simply

haven't discovered it yet. We can use Theorem 16.6 to "bootstrap" our original "good but not great" 3SAT algorithm to discover the optimal one. The idea is that we can phrase the question of whether there exists a size s circuit that computes 3SAT for all length n inputs as follows: *there exists* a size $\leq s$ circuit C such that *for every* formula φ described by a string of length at most n, if $C(\varphi) = 1$ then *there exists* an assignment x to the variables of φ that satisfies it. One can see that this is a statement of the form (16.2) and hence if $\mathbf{P} = \mathbf{NP}$ we can solve it in polynomial time as well. We can therefore imagine investing huge computational resources in running A one time to discover the circuit C^* and then using C^* for all further computation.

16.5 APPROXIMATING COUNTING PROBLEMS AND POSTERIOR SAMPLING (ADVANCED, OPTIONAL)

Given a Boolean circuit C, if $\mathbf{P} = \mathbf{NP}$ then we can find an input x (if one exists) such that $C(x) = 1$. But what if there is more than one x like that? Clearly we can't efficiently output all such x's; there might be exponentially many. But we can get an arbitrarily good multiplicative approximation (i.e., a $1 \pm \epsilon$ factor for arbitrarily small $\epsilon > 0$) for the number of such x's, as well as output a (nearly) uniform member of this set. The details are beyond the scope of this book, but this result is formally stated in the following theorem (whose proof is omitted).

Theorem 16.7 — Approximate counting if P = NP. Let $V : \{0,1\}^* \to \{0,1\}$ be some polynomial-time algorithm, and suppose that $\mathbf{P} = \mathbf{NP}$. Then there exists an algorithm $COUNT_V$ that on input $x, 1^m, \epsilon$, runs in time polynomial in $|x|, m, 1/\epsilon$ and outputs a number in $[2^m + 1]$ satisfying

$$(1-\epsilon)COUNT_V(x,m,\epsilon) \leq \left|\{y \in \{0,1\}^m : V(xy) = 1\}\right| \leq (1+\epsilon)COUNT_V(x,m,\epsilon) .$$

In other words, the algorithm $COUNT_V$ gives an approximation up to a factor of $1 \pm \epsilon$ for the number of *witnesses* for x with respect to the verifying algorithm V. Once again, to understand this theorem it can be useful to see how it implies that if $\mathbf{P} = \mathbf{NP}$ then there is a polynomial-time algorithm that given a graph G and a number k, can compute a number K that is within a 1 ± 0.01 factor equal to the number of simple paths in G of length k. (That is, K is between 0.99 to 1.01 times the number of such paths.)

Posterior sampling and probabilistic programming. The algorithm for counting can also be extended to *sampling* from a given posterior distribution. That is, if $C : \{0,1\}^n \to \{0,1\}^m$ is a Boolean circuit and

$y \in \{0,1\}^m$, then if **P** $=$ **NP** we can sample from (a close approximation of) the distribution of uniform $x \in \{0,1\}^n$ conditioned on $C(x) = y$. This task is known as *posterior sampling* and is crucial for Bayesian data analysis. These days it is known how to achieve posterior sampling only for circuits C of very special form, and even in these cases more often than not we do have guarantees on the quality of the sampling algorithm. The field of making inferences by sampling from posterior distribution specified by circuits or programs is known as probabilistic programming.

16.6 WHAT DOES ALL OF THIS IMPLY?

So, what will happen if we have a $10^6 n$ algorithm for $3SAT$? We have mentioned that **NP**-hard problems arise in many contexts, and indeed scientists, engineers, programmers and others routinely encounter such problems in their daily work. A better $3SAT$ algorithm will probably make their lives easier, but that is the wrong place to look for the most foundational consequences. Indeed, while the invention of electronic computers did of course make it easier to do calculations that people were already doing with mechanical devices and pen and paper, the main applications computers are used for today were not even imagined before their invention.

An exponentially faster algorithm for all **NP** problems would be no less radical an improvement (and indeed, in some sense would be more) than the computer itself, and it is as hard for us to imagine what it would imply as it was for Babbage to envision today's world. For starters, such an algorithm would completely change the way we program computers. Since we could automatically find the "best" (in any measure we chose) program that achieves a certain task, we would not need to define *how* to achieve a task, but only specify tests as to what would be a good solution, and could also ensure that a program satisfies an exponential number of tests without actually running them.

The possibility that **P** $=$ **NP** is often described as "automating creativity". There is something to that analogy, as we often think of a creative solution as one that is hard to discover but that, once the "spark" hits, is easy to verify. But there is also an element of hubris to that statement, implying that the most impressive consequence of such an algorithmic breakthrough will be that computers would succeed in doing something that humans already do today. Nevertheless, artificial intelligence, like many other fields, will clearly be greatly impacted by an efficient 3SAT algorithm. For example, it is clearly much easier to find a better Chess-playing algorithm when, given any algorithm P, you can find the smallest algorithm P' that plays Chess better than P. Moreover, as we mentioned above, much of machine

learning (and statistical reasoning in general) is about finding "simple" concepts that explain the observed data, and if **NP** = **P**, we could search for such concepts automatically for any notion of "simplicity" we see fit. In fact, we could even "skip the middle man" and do an automatic search for the learning algorithm with smallest generalization error. Ultimately the field of Artificial Intelligence is about trying to "shortcut" billions of years of evolution to obtain artificial programs that match (or beat) the performance of natural ones, and a fast algorithm for **NP** would provide the ultimate shortcut.[8]

More generally, a faster algorithm for **NP** problems would be immensely useful in any field where one is faced with computational or quantitative problems— which is basically all fields of science, math, and engineering. This will not only help with concrete problems such as designing a better bridge, or finding a better drug, but also with addressing basic mysteries such as trying to find scientific theories or "laws of nature". In a fascinating talk, physicist Nima Arkani-Hamed discusses the effort of finding scientific theories in much the same language as one would describe solving an **NP** problem, for which the solution is easy to verify or seems "inevitable", once found, but that requires searching through a huge landscape of possibilities to reach, and that often can get "stuck" at local optima:

> *"the laws of nature have this amazing feeling of inevitability... which is associated with local perfection."*
>
> *"The classical picture of the world is the top of a local mountain in the space of ideas. And you go up to the top and it looks amazing up there and absolutely incredible. And you learn that there is a taller mountain out there. Find it, Mount Quantum.... they're not smoothly connected ... you've got to make a jump to go from classical to quantum ... This also tells you why we have such major challenges in trying to extend our understanding of physics. We don't have these knobs, and little wheels, and twiddles that we can turn. We have to learn how to make these jumps. And it is a tall order. And that's why things are difficult."*

Finding an efficient algorithm for **NP** amounts to always being able to search through an exponential space and find not just the "local" mountain, but the tallest peak.

But perhaps more than any computational speedups, a fast algorithm for **NP** problems would bring about a *new type of understanding*. In many of the areas where **NP**-completeness arises, it is not as much a barrier for solving computational problems as it is a barrier for obtaining "closed-form formulas" or other types of more constructive descriptions of the behavior of natural, biological, social and other systems. A better algorithm for **NP**, even if it is "merely" $2^{\sqrt{n}}$-time, seems to require obtaining a new way to understand these types of systems, whether it is characterizing Nash equilibria, spin-glass configurations,

[8] One interesting theory is that **P** = **NP** and evolution has already discovered this algorithm, which we are already using without realizing it. At the moment, there seems to be very little evidence for such a scenario. In fact, we have some partial results in the other direction showing that, regardless of whether **P** = **NP**, many types of "local search" or "evolutionary" algorithms require exponential time to solve 3SAT and other **NP**-hard problems.

entangled quantum states, or any of the other questions where **NP** is currently a barrier for analytical understanding. Such new insights would be very fruitful regardless of their computational utility.

Big Idea 23 If **P** $=$ **NP**, we can efficiently solve a fantastic number of decision, search, optimization, counting, and sampling problems from all areas of human endeavors.

16.7 CAN P $\neq$ NP BE NEITHER TRUE NOR FALSE?

The Continuum Hypothesis is a conjecture made by Georg Cantor in 1878, positing the non-existence of a certain type of infinite cardinality. (One way to phrase it is that for every infinite subset S of the real numbers $\mathbb{R}$, either there is a one-to-one and onto function $f : S \rightarrow \mathbb{R}$ or there is a one-to-one and onto function $f : S \rightarrow \mathbb{N}$.) This was considered one of the most important open problems in set theory, and settling its truth or falseness was the first problem put forward by Hilbert in the 1900 address we mentioned before. However, using the theories developed by Gödel and Turing, in 1963 Paul Cohen proved that both the Continuum Hypothesis and its negation are consistent with the standard axioms of set theory (i.e., the Zermelo-Fraenkel axioms + the Axiom of choice, or "ZFC" for short). Formally, what he proved is that if ZFC is consistent, then so is ZFC when we assume either the continuum hypothesis or its negation.

Today, many (though not all) mathematicians interpret this result as saying that the Continuum Hypothesis is neither true nor false, but rather is an axiomatic choice that we are free to make one way or the other. Could the same hold for **P** $\neq$ **NP**?

In short, the answer is *No*. For example, suppose that we are trying to decide between the "3SAT is easy" conjecture (there is an $10^6 n$ time algorithm for 3SAT) and the "3SAT is hard" conjecture (for every n, any NAND-CIRC program that solves n variable 3SAT takes $2^{10^{-6}n}$ lines). Then, since for $n = 10^8$, $2^{10^{-6}n} > 10^6 n$, this boils down to the finite question of deciding whether or not there is a 10^{13}-line NAND-CIRC program deciding 3SAT on formulas with 10^8 variables. If there is such a program then there is a finite proof of its existence, namely the approximately 1TB file describing the program, and for which the verification is the (finite in principle though infeasible in practice) process of checking that it succeeds on all inputs.[9] If there isn't such a program, then there is also a finite proof of that, though any such proof would take longer since we would need to enumerate over all *programs* as well. Ultimately, since it boils down to a finite statement about bits and numbers; either the statement or its negation

[9] This inefficiency is not necessarily inherent. Later in this course we may discuss results in program-checking, interactive proofs, and average-case complexity, that can be used for efficient verification of proofs of related statements. In contrast, the inefficiency of verifying *failure* of all programs could well be inherent.

must follow from the standard axioms of arithmetic in a finite number of arithmetic steps. Thus, we cannot justify our ignorance in distinguishing between the "3SAT easy" and "3SAT hard" cases by claiming that this might be an inherently ill-defined question. Similar reasoning (with different numbers) applies to other variants of the **P** vs **NP** question. We note that in the case that 3SAT is hard, it may well be that there is no *short* proof of this fact using the standard axioms, and this is a question that people have been studying in various restricted forms of proof systems.

16.8 IS $\mathrm{P} = \mathrm{NP}$ "IN PRACTICE"?

The fact that a problem is **NP**-hard means that we believe there is no efficient algorithm that solves it in the *worst case*. It does not, however, mean that every single instance of the problem is hard. For example, if all the clauses in a 3SAT instance φ contain the same variable x_i (possibly in negated form), then by guessing a value to x_i we can reduce φ to a 2SAT instance which can then be efficiently solved. Generalizations of this simple idea are used in "SAT solvers", which are algorithms that have solved certain specific interesting SAT formulas with thousands of variables, despite the fact that we believe SAT to be exponentially hard in the worst case. Similarly, a lot of problems arising in economics and machine learning are **NP**-hard.[10] And yet vendors and customers manage to figure out market-clearing prices (as economists like to point out, there is milk on the shelves) and mice succeed in distinguishing cats from dogs. Hence people (and machines) seem to regularly succeed in solving interesting instances of **NP**-hard problems, typically by using some combination of guessing while making local improvements.

[10] Actually, the computational difficulty of problems in economics such as finding optimal (or any) equilibria is quite subtle. Some variants of such problems are **NP**-hard, while others have a certain "intermediate" complexity.

It is also true that there are many interesting instances of **NP**-hard problems that we do *not* currently know how to solve. Across all application areas, whether it is scientific computing, optimization, control or more, people often encounter hard instances of **NP** problems on which our current algorithms fail. In fact, as we will see, all of our digital security infrastructure relies on the fact that some concrete and easy-to-generate instances of, say, 3SAT (or, equivalently, any other **NP**-hard problem) are exponentially hard to solve.

Thus it would be wrong to say that **NP** is easy "in practice", nor would it be correct to take **NP**-hardness as the "final word" on the complexity of a problem, particularly when we have more information about how any given instance is generated. Understanding both the "typical complexity" of **NP** problems, as well as the power and limitations of certain heuristics (such as various local-search based algorithms) is a very active area of research. We will see more on these topics later in this course.

11

16.9 WHAT IF $P \neq NP$?

So, **P** $=$ **NP** would give us all kinds of fantastical outcomes. But we strongly suspect that **P** $\neq$ **NP**, and moreover that there is no much-better-than-brute-force algorithm for 3SAT. If indeed that is the case, is it all bad news?

One might think that impossibility results, telling you that you *cannot* do something, is the kind of cloud that does not have a silver lining. But in fact, as we already alluded to before, it does. A hard (in a sufficiently strong sense) problem in **NP** can be used to create a code that *cannot be broken*, a task that for thousands of years has been the dream of not just spies but of many scientists and mathematicians over the generations. But the complexity viewpoint turned out to yield much more than simple codes, achieving tasks that people had previously not even dared to dream of. These include the notion of *public key cryptography*, allowing two people to communicate securely without ever having exchanged a secret key; *electronic cash*, allowing private and secure transaction without a central authority; and *secure multiparty computation*, enabling parties to compute a joint function on private inputs without revealing any extra information about it. Also, as we will see, computational hardness can be used to replace the role of *randomness* in many settings.

Furthermore, while it is often convenient to pretend that computational problems are simply handed to us, and that our job as computer scientists is to find the most efficient algorithm for them, this is not how things work in most computing applications. Typically even formulating the problem to solve is a highly non-trivial task. When we discover that the problem we want to solve is **NP**-hard, this might be a useful sign that we used the wrong formulation for it.

Beyond all these, the quest to understand computational hardness — including the discoveries of lower bounds for restricted computational models, as well as new types of reductions (such as those arising from "probabilistically checkable proofs") — has already had surprising *positive* applications to problems in algorithm design, as well as in coding for both communication and storage. This is not surprising since, as we mentioned before, from group theory to the theory of relativity, the pursuit of impossibility results has often been one of the most fruitful enterprises of mankind.

Chapter Recap

- The question of whether **P** $=$ **NP** is one of the most important and fascinating questions of com-

[11] Talk more about coping with NP hardness. Main two approaches are *heuristics* such as SAT solvers that succeed on *some* instances, and *proxy measures* such as mathematical relaxations that instead of solving problem X (e.g., an integer program) solve program X' (e.g., a linear program) that is related to that. Maybe give compressed sensing as an example, and least square minimization as a proxy for maximum apostoriori probability.

puter science and science at large, touching on all fields of the natural and social sciences, as well as mathematics and engineering.

- Our current evidence and understanding supports the "SAT hard" scenario that there is no much-better-than-brute-force algorithm for 3SAT or many other **NP**-hard problems.
- We are very far from *proving* this, however. Researchers have studied proving lower bounds on the number of gates to compute explicit functions in *restricted forms* of circuits, and have made some advances in this effort, along the way generating mathematical tools that have found other uses. However, we have made essentially no headway in proving lower bounds for *general* models of computation such as Boolean circuits and Turing machines. Indeed, we currently do not even know how to rule out the possibility that for every $n \in \mathbb{N}$, *SAT* restricted to n-length inputs has a Boolean circuit of less than $10n$ gates (even though there *exist* n-input functions that require at least $2^n/(10n)$ gates to compute).
- Understanding how to cope with this computational intractability, and even benefit from it, comprises much of the research in theoretical computer science.

16.10 EXERCISES

16.11 BIBLIOGRAPHICAL NOTES

As mentioned before, Aaronson's survey [Aar16] is a great exposition of the **P** vs **NP** problem. Another recommended survey by Aaronson is [Aar05] which discusses the question of whether **NP** complete problems could be computed by any physical means.

The paper [BU11] discusses some results about problems in the polynomial hierarchy.

17
Space bounded computation

PLAN: Example of space bounded algorithms, importance of preserving space. The classes L and PSPACE, space hierarchy theorem, PSPACE=NPSPACE, constant space = regular languages.

17.1 EXERCISES

17.2 BIBLIOGRAPHICAL NOTES

IV

RANDOMIZED COMPUTATION

Learning Objectives:

- Review the basic notion of probability theory that we will use.
- Sample spaces, and in particular the space $\{0,1\}^n$
- Events, probabilities of unions and intersections.
- Random variables and their expectation, variance, and standard deviation.
- Independence and correlation for both events and random variables.
- Markov, Chebyshev and Chernoff tail bounds (bounding the probability that a random variable will deviate from its expectation).

18
Probability Theory 101

"God doesn't play dice with the universe", Albert Einstein

"Einstein was doubly wrong ... not only does God definitely play dice, but He sometimes confuses us by throwing them where they can't be seen.", Stephen Hawking

" 'The probability of winning a battle has no place in our theory because it does not belong to any [random experiment]. Probability cannot be applied to this problem any more than the physical concept of work can be applied to the 'work' done by an actor reciting his part.", Richard Von Mises, 1928 (paraphrased)

"I am unable to see why 'objectivity' requires us to interpret every probability as a frequency in some random experiment; particularly when in most problems probabilities are frequencies only in an imaginary universe invented just for the purpose of allowing a frequency interpretation.", E.T. Jaynes, 1976

Before we show how to use randomness in algorithms, let us do a quick review of some basic notions in probability theory. This is not meant to replace a course on probability theory, and if you have not seen this material before, I highly recommend you look at additional resources to get up to speed. Fortunately, we will not need many of the advanced notions of probability theory, but, as we will see, even the so-called "simple" setting of tossing n coins can lead to very subtle and interesting issues.

This chapter: A non-mathy overview

This chapter contains an overview of the basics of probability theory, as needed for understanding randomized computation. The main topics covered are the notions of:

1. A *sample space*, which for us will almost always consist of the set of all possible outcomes of the experiment of tossing a finite number of independent coins.

2. An *event*, which is simply a subset of the sample space, with the probability of the event happening being the fraction of outcomes that are in this subset.

3. A *random variable*, which is a way to assign some number or statistic to an outcome of the sample space.

4. The notion of *conditioning*, which corresponds to how the value of a random variable (or the probability of an event) changes if we restrict attention to outcomes for which the value of another variable is known (or for which some other event has happened). Random variables and events that have no impact on one another are called *independent*.

5. *Expectation*, which is the average of a random variable, and *concentration bounds* which quantify the probability that a random variable can "stray too far" from its expected value.

These concepts are at once both basic and subtle. While we will not need many "fancy" topics covered in statistics courses, including special distributions (e.g., gemoetric, Poisson, exponential, Gaussian, etc.), nor topics such as hypothesis testing or regression, this doesn't mean that the probability we use is "trivial". The human brain has not evolved to do probabilistic reasoning very well, and notions such as conditioning and independence can be quite subtle and confusing even in the basic setting of tossing a random coin. However, this is all the more reason that studying these notions in this basic setting is useful not just for following this book, but also as a strong foundation for "fancier topics".

18.1 RANDOM COINS

The nature of randomness and probability is a topic of great philosophical, scientific and mathematical depth. Is there actual randomness in the world, or does it proceed in a deterministic clockwork fashion from some initial conditions set at the beginning of time? Does probability refer to our uncertainty of beliefs, or to the frequency of occurrences in repeated experiments? How can we define probability over infinite sets?

These are all important questions that have been studied and debated by scientists, mathematicians, statisticians and philosophers. Fortunately, we will not need to deal directly with these questions here. We will be mostly interested in the setting of tossing n random,

unbiased and independent coins. Below we define the basic probabilistic objects of *events* and *random variables* when restricted to this setting. These can be defined for much more general probabilistic experiments or *sample spaces*, and later on we will briefly discuss how this can be done. However, the n-coin case is sufficient for almost everything we'll need in this course.

If instead of "heads" and "tails" we encode the sides of each coin by "zero" and "one", we can encode the result of tossing n coins as a string in $\{0,1\}^n$. Each particular outcome $x \in \{0,1\}^n$ is obtained with probability 2^{-n}. For example, if we toss three coins, then we obtain each of the 8 outcomes $000, 001, 010, 011, 100, 101, 110, 111$ with probability $2^{-3} = 1/8$ (see also Fig. 18.1). We can describe the experiment of tossing n coins as choosing a string x uniformly at random from $\{0,1\}^n$, and hence we'll use the shorthand $x \sim \{0,1\}^n$ for x that is chosen according to this experiment.

An *event* is simply a subset A of $\{0,1\}^n$. The *probability of* A, denoted by $\Pr_{x\sim\{0,1\}^n}[A]$ (or $\Pr[A]$ for short, when the sample space is understood from the context), is the probability that an x chosen uniformly at random will be contained in A. Note that this is the same as $|A|/2^n$ (where $|A|$ as usual denotes the number of elements in the set A). For example, the probability that x has an even number of ones is $\Pr[A]$ where $A = \{x : \sum_{i=0}^{n-1} x_i = 0 \mod 2\}$. In the case $n = 3$, $A = \{000, 011, 101, 110\}$, and hence $\Pr[A] = \frac{4}{8} = \frac{1}{2}$ (see Fig. 18.2). It turns out this is true for every n:

Lemma 18.1 For every $n > 0$,

$$\Pr_{x\sim\{0,1\}^n}[\sum_{i=0}^{n-1} x_i \text{ is even }] = 1/2$$

P To test your intuition on probability, try to stop here and prove the lemma on your own.

Proof of Lemma 18.1. We prove the lemma by induction on n. For the case $n = 1$ it is clear since $x = 0$ is even and $x = 1$ is odd, and hence the probability that $x \in \{0,1\}$ is even is $1/2$. Let $n > 1$. We assume by induction that the lemma is true for $n-1$ and we will prove it for n. We split the set $\{0,1\}^n$ into four disjoint sets E_0, E_1, O_0, O_1, where for $b \in \{0,1\}$, E_b is defined as the set of $x \in \{0,1\}^n$ such that $x_0 \cdots x_{n-2}$ has even number of ones and $x_{n-1} = b$ and similarly O_b is the set of $x \in \{0,1\}^n$ such that $x_0 \cdots x_{n-2}$ has odd number of ones and $x_{n-1} = b$. Since E_0 is obtained by simply extending $n-1$-length string with even number of ones by the digit 0, the size of E_0 is simply the

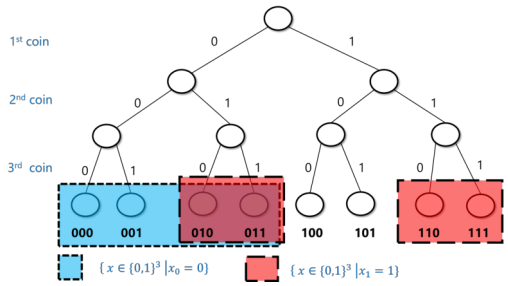


Figure 18.1: The probabilistic experiment of tossing three coins corresponds to making $2 \times 2 \times 2 = 8$ choices, each with equal probability. In this example, the blue set corresponds to the event $A = \{x \in \{0,1\}^3 \mid x_0 = 0\}$ where the first coin toss is equal to 0, and the pink set corresponds to the event $B = \{x \in \{0,1\}^3 \mid x_1 = 1\}$ where the second coin toss is equal to 1 (with their intersection having a purplish color). As we can see, each of these events contains 4 elements (out of 8 total) and so has probability $1/2$. The intersection of A and B contains two elements, and so the probability that both of these events occur is $2/8 = 1/4$.

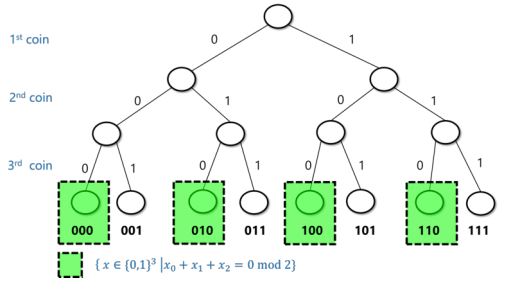


Figure 18.2: The event that if we toss three coins $x_0, x_1, x_2 \in \{0,1\}$ then the sum of the x_i's is even has probability $1/2$ since it corresponds to exactly 4 out of the 8 possible strings of length 3.

number of such $n-1$-length strings which by the induction hypothesis is $2^{n-1}/2 = 2^{n-2}$. The same reasoning applies for E_1, O_0, and O_1. Hence each one of the four sets E_0, E_1, O_0, O_1 is of size 2^{n-2}. Since $x \in \{0,1\}^n$ has an even number of ones if and only if $x \in E_0 \cup O_1$ (i.e., either the first $n-1$ coordinates sum up to an even number and the final coordinate is 0 or the first $n-1$ coordinates sum up to an odd number and the final coordinate is 1), we get that the probability that x satisfies this property is

$$\frac{|E_0 \cup O_1|}{2^n} = \frac{2^{n-2} + 2^{n-2}}{2^n} = \frac{1}{2} ,$$

using the fact that E_0 and O_1 are disjoint and hence $|E_0 \cup O_1| = |E_0| + |O_1|$.

■

We can also use the *intersection* ($\cap$) and *union* ($\cup$) operators to talk about the probability of both event A *and* event B happening, or the probability of event A *or* event B happening. For example, the probability p that x has an *even* number of ones *and* $x_0 = 1$ is the same as $\Pr[A \cap B]$ where $A = \{x \in \{0,1\}^n : \sum_{i=0}^{n-1} x_i = 0 \mod 2\}$ and $B = \{x \in \{0,1\}^n : x_0 = 1\}$. This probability is equal to $1/4$ for $n > 1$. (It is a great exercise for you to pause here and verify that you understand why this is the case.)

Because intersection corresponds to considering the logical AND of the conditions that two events happen, while union corresponds to considering the logical OR, we will sometimes use the $\wedge$ and $\vee$ operators instead of $\cap$ and $\cup$, and so write this probability $p = \Pr[A \cap B]$ defined above also as

$$\Pr_{x \sim \{0,1\}^n} \left[\sum_i x_i = 0 \mod 2 \wedge x_0 = 1 \right] .$$

If $A \subseteq \{0,1\}^n$ is an event, then $\overline{A} = \{0,1\}^n \setminus A$ corresponds to the event that A does *not* happen. Since $|\overline{A}| = 2^n - |A|$, we get that

$$\Pr[\overline{A}] = \frac{|\overline{A}|}{2^n} = \frac{2^n - |A|}{2^n} = 1 - \frac{|A|}{2^n} = 1 - \Pr[A]$$

This makes sense: since A happens if and only if $\overline{A}$ does *not* happen, the probability of $\overline{A}$ should be one minus the probability of A.

R

Remark 18.2 — Remember the sample space. While the above definition might seem very simple and almost trivial, the human mind seems not to have evolved for probabilistic reasoning, and it is surprising how often people can get even the simplest settings of probability wrong. One way to make sure you don't get confused

when trying to calculate probability statements is to always ask yourself the following two questions: (**1**) Do I understand what is the **sample space** that this probability is taken over?, and (**2**) Do I understand what is the definition of the **event** that we are analyzing?.

For example, suppose that I were to randomize seating in my course, and then it turned out that students sitting in row 7 performed better on the final: how surprising should we find this? If we started out with the hypothesis that there is something special about the number 7 and chose it ahead of time, then the event that we are discussing is the event A that students sitting in number 7 had better performance on the final, and we might find it surprising. However, if we first looked at the results and then chose the row whose average performance is best, then the event we are discussing is the event B that there exists *some* row where the performance is higher than the overall average. B is a superset of A, and its probability (even if there is no correlation between sitting and performance) can be quite significant.

18.1.1 Random variables

Events correspond to Yes/No questions, but often we want to analyze finer questions. For example, if we make a bet at the roulette wheel, we don't want to just analyze whether we won or lost, but also *how much* we've gained. A (real valued) *random variable* is simply a way to associate a number with the result of a probabilistic experiment. Formally, a random variable is a function $X : \{0,1\}^n \to \mathbb{R}$ that maps every outcome $x \in \{0,1\}^n$ to an element $X(x) \in \mathbb{R}$. For example, the function $SUM : \{0,1\}^n \to \mathbb{R}$ that maps x to the sum of its coordinates (i.e., to $\sum_{i=0}^{n-1} x_i$) is a random variable.

The *expectation* of a random variable X, denoted by $\mathbb{E}[X]$, is the average value that this number takes, taken over all draws from the probabilistic experiment. In other words, the expectation of X is defined as follows:

$$\mathbb{E}[X] = \sum_{x \in \{0,1\}^n} 2^{-n} X(x) \ .$$

If X and Y are random variables, then we can define $X + Y$ as simply the random variable that maps a point $x \in \{0,1\}^n$ to $X(x) + Y(x)$. One basic and very useful property of the expectation is that it is *linear*:

Lemma 18.3 — Linearity of expectation.

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

Proof.

$$\mathbb{E}[X+Y] = \sum_{x\in\{0,1\}^n} 2^{-n}\left(X(x)+Y(x)\right) =$$
$$\sum_{x\in\{0,1\}^n} 2^{-n}X(x) + \sum_{x\in\{0,1\}^n} 2^{-n}Y(x) =$$
$$\mathbb{E}[X]+\mathbb{E}[Y]$$

■

Similarly, $\mathbb{E}[kX] = k\,\mathbb{E}[X]$ for every $k \in \mathbb{R}$.

Solved Exercise 18.1 — Expectation of sum. Let $X : \{0,1\}^n \to \mathbb{R}$ be the random variable that maps $x \in \{0,1\}^n$ to $x_0 + x_1 + ... + x_{n-1}$. Prove that $\mathbb{E}[X] = n/2$.

■

Solution:

We can solve this using the linearity of expectation. We can define random variables $X_0, X_1, \ldots, X_{n-1}$ such that $X_i(x) = x_i$. Since each x_i equals 1 with probability $1/2$ and 0 with probability $1/2$, $\mathbb{E}[X_i] = 1/2$. Since $X = \sum_{i=0}^{n-1} X_i$, by the linearity of expectation

$$\mathbb{E}[X] = \mathbb{E}[X_0] + \mathbb{E}[X_1] + \cdots + \mathbb{E}[X_{n-1}] = \tfrac{n}{2} .$$

■

If you have not seen discrete probability before, please go over this argument again until you are sure you follow it; it is a prototypical simple example of the type of reasoning we will employ again and again in this course.

If A is an event, then 1_A is the random variable such that $1_A(x)$ equals 1 if $x \in A$, and $1_A(x) = 0$ otherwise. Note that $\Pr[A] = \mathbb{E}[1_A]$ (can you see why?). Using this and the linearity of expectation, we can show one of the most useful bounds in probability theory:

Lemma 18.4 — Union bound. For every two events A, B, $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$

Before looking at the proof, try to see why the union bound makes intuitive sense. We can also prove it directly from the definition of probabilities and the cardinality of sets, together with the equation $|A \cup B| \leq |A| + |B|$. Can you see why the latter equation is true? (See also Fig. 18.3.)

Proof of Lemma 18.4. For every x, the variable $1_{A\cup B}(x) \leq 1_A(x)+1_B(x)$. Hence, $\Pr[A\cup B] = \mathbb{E}[1_{A\cup B}] \leq \mathbb{E}[1_A+1_B] = \mathbb{E}[1_A]+\mathbb{E}[1_B] = \Pr[A]+\Pr[B]$. ■

The way we often use this in theoretical computer science is to argue that, for example, if there is a list of 100 bad events that can happen, and each one of them happens with probability at most $1/10000$, then with probability at least $1 - 100/10000 = 0.99$, no bad event happens.

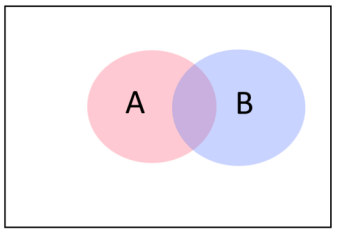


Figure 18.3: The *union bound* tells us that the probability of A or B happening is at most the sum of the individual probabilities. We can see it by noting that for every two sets $|A \cup B| \leq |A| + |B|$ (with equality only if A and B have no intersection).

18.1.2 Distributions over strings

While most of the time we think of random variables as having as output a *real number*, we sometimes consider random variables whose output is a *string*. That is, we can think of a map $Y : \{0,1\}^n \to \{0,1\}^*$ and consider the "random variable" Y such that for every $y \in \{0,1\}^*$, the probability that Y outputs y is equal to $\frac{1}{2^n}|\{x \in \{0,1\}^n \mid Y(x) = y\}|$. To avoid confusion, we will typically refer to such string-valued random variables as *distributions* over strings. So, a *distribution* Y over strings $\{0,1\}^*$ can be thought of as a finite collection of strings $y_0, \ldots, y_{M-1} \in \{0,1\}^*$ and probabilities $p_0, \ldots, p_{M-1}$ (which are non-negative numbers summing up to one), so that $\Pr[Y = y_i] = p_i$.

Two distributions Y and Y' are *identical* if they assign the same probability to every string. For example, consider the following two functions $Y, Y' : \{0,1\}^2 \to \{0,1\}^2$. For every $x \in \{0,1\}^2$, we define $Y(x) = x$ and $Y'(x) = x_0(x_0 \oplus x_1)$ where $\oplus$ is the XOR operations. Although these are two different functions, they induce the same distribution over $\{0,1\}^2$ when invoked on a uniform input. The distribution $Y(x)$ for $x \sim \{0,1\}^2$ is of course the uniform distribution over $\{0,1\}^2$. On the other hand Y' is simply the map $00 \mapsto 00$, $01 \mapsto 01$, $10 \mapsto 11$, $11 \mapsto 10$ which is a permutation of Y.

18.1.3 More general sample spaces

While throughout most of this book we assume that the underlying probabilistic experiment corresponds to tossing n independent coins, all the claims we make easily generalize to sampling x from a more general finite or countable set S (and not-so-easily generalizes to uncountable sets S as well). A *probability distribution* over a finite set S is simply a function $\mu : S \to [0,1]$ such that $\sum_{x\in S} \mu(x) = 1$. We think of this as the experiment where we obtain every $x \in S$ with probability $\mu(x)$, and sometimes denote this as $x \sim \mu$. In particular, tossing n random coins corresponds to the probability distribution $\mu : \{0,1\}^n \to [0,1]$ defined as $\mu(x) = 2^{-n}$ for every $x \in \{0,1\}^n$. An *event* A is a subset of S, and the probability of A, which we denote by

$\Pr_\mu[A]$, is $\sum_{x\in A}\mu(x)$. A *random variable* is a function $X : S \to \mathbb{R}$, where the probability that $X = y$ is equal to $\sum_{x\in S \text{ s.t. } X(x)=y}\mu(x)$.

18.2 CORRELATIONS AND INDEPENDENCE

One of the most delicate but important concepts in probability is the notion of *independence* (and the opposing notion of *correlations*). Subtle correlations are often behind surprises and errors in probability and statistical analysis, and several mistaken predictions have been blamed on miscalculating the correlations between, say, housing prices in Florida and Arizona, or voter preferences in Ohio and Michigan. See also Joe Blitzstein's aptly named talk "Conditioning is the Soul of Statistics". (Another thorny issue is of course the difference between *correlation* and *causation*. Luckily, this is another point we don't need to worry about in our clean setting of tossing n coins.)

Two events A and B are *independent* if the fact that A happens makes B neither more nor less likely to happen. For example, if we think of the experiment of tossing 3 random coins $x \in \{0,1\}^3$, and we let A be the event that $x_0 = 1$ and B the event that $x_0 + x_1 + x_2 \geq 2$, then if A happens it is more likely that B happens, and hence these events are *not* independent. On the other hand, if we let C be the event that $x_1 = 1$, then because the second coin toss is not affected by the result of the first one, the events A and C are independent.

The formal definition is that events A and B are *independent* if $\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$. If $\Pr[A \cap B] > \Pr[A] \cdot \Pr[B]$ then we say that A and B are *positively correlated*, while if $\Pr[A \cap B] < \Pr[A] \cdot \Pr[B]$ then we say that A and B are *negatively correlated* (see Fig. 18.4).

If we consider the above examples on the experiment of choosing $x \in \{0,1\}^3$ then we can see that

$$\Pr[x_0 = 1] = \tfrac{1}{2}$$

$$\Pr[x_0 + x_1 + x_2 \geq 2] = \Pr[\{011, 101, 110, 111\}] = \tfrac{4}{8} = \tfrac{1}{2}$$

but

$$\Pr[x_0 = 1 \ \wedge \ x_0 + x_1 + x_2 \geq 2] = \Pr[\{101, 110, 111\}] = \tfrac{3}{8} > \tfrac{1}{2} \cdot \tfrac{1}{2}$$

and hence, as we already observed, the events $\{x_0 = 1\}$ and $\{x_0 + x_1 + x_2 \geq 2\}$ are not independent and in fact are positively correlated. On the other hand, $\Pr[x_0 = 1 \wedge x_1 = 1] = \Pr[\{110, 111\}] = \frac{2}{8} = \frac{1}{2} \cdot \frac{1}{2}$ and hence the events $\{x_0 = 1\}$ and $\{x_1 = 1\}$ are indeed independent.

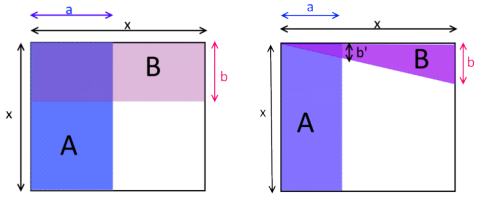


Figure 18.4: Two events A and B are *independent* if $\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$. In the two figures above, the empty $x \times x$ square is the sample space, and A and B are two events in this sample space. In the left figure, A and B are independent, while in the right figure they are negatively correlated, since B is less likely to occur if we condition on A (and vice versa). Mathematically, one can see this by noticing that in the left figure the areas of A and B respectively are $a \cdot x$ and $b \cdot x$, and so their probabilities are $\frac{a\cdot x}{x^2} = \frac{a}{x}$ and $\frac{b\cdot x}{x^2} = \frac{b}{x}$ respectively, while the area of $A \cap B$ is $a \cdot b$ which corresponds to the probability $\frac{a\cdot b}{x^2}$. In the right figure, the area of the triangle B is $\frac{b\cdot x}{2}$ which corresponds to a probability of $\frac{b}{2x}$, but the area of $A \cap B$ is $\frac{b'\cdot a}{2}$ for some $b' < b$. This means that the probability of $A \cap B$ is $\frac{b'\cdot a}{2x^2} < \frac{b}{2x} \cdot \frac{a}{x}$, or in other words $\Pr[A \cap B] < \Pr[A] \cdot \Pr[B]$.

R

Remark 18.5 — Disjointness vs independence. People sometimes confuse the notion of *disjointness* and *in-*

dependence, but these are actually quite different. Two events A and B are *disjoint* if $A \cap B = \emptyset$, which means that if A happens then B definitely does not happen. They are *independent* if $\Pr[A \cap B] = \Pr[A]\Pr[B]$ which means that knowing that A happens gives us no information about whether B happened or not. If A and B have non-zero probability, then being disjoint implies that they are *not* independent, since in particular it means that they are negatively correlated.

Conditional probability: If A and B are events, and A happens with non-zero probability then we define the probability that B happens *conditioned on* A to be $\Pr[B|A] = \Pr[A \cap B]/\Pr[A]$. This corresponds to calculating the probability that B happens if we already know that A happened. Note that A and B are independent if and only if $\Pr[B|A] = \Pr[B]$.

More than two events: We can generalize this definition to more than two events. We say that events $A_1, \ldots, A_k$ are *mutually independent* if knowing that any set of them occurred or didn't occur does not change the probability that an event outside the set occurs. Formally, the condition is that for every subset $I \subseteq [k]$,

$$\Pr[\wedge_{i \in I} A_i] = \prod_{i \in I} \Pr[A_i].$$

For example, if $x \sim \{0,1\}^3$, then the events $\{x_0 = 1\}$, $\{x_1 = 1\}$ and $\{x_2 = 1\}$ are mutually independent. On the other hand, the events $\{x_0 = 1\}$, $\{x_1 = 1\}$ and $\{x_0 + x_1 = 0 \mod 2\}$ are *not* mutually independent, even though every pair of these events is independent (can you see why? see also Fig. 18.5).

18.2.1 Independent random variables

We say that two random variables $X : \{0,1\}^n \to \mathbb{R}$ and $Y : \{0,1\}^n \to \mathbb{R}$ are independent if for every $u, v \in \mathbb{R}$, the events $\{X = u\}$ and $\{Y = v\}$ are independent. (We use $\{X = u\}$ as shorthand for $\{x \mid X(x) = u\}$.) In other words, X and Y are independent if $\Pr[X = u \wedge Y = v] = \Pr[X = u]\Pr[Y = v]$ for every $u, v \in \mathbb{R}$. For example, if two random variables depend on the result of tossing different coins then they are independent:

Lemma 18.6 Suppose that $S = \{s_0, \ldots, s_{k-1}\}$ and $T = \{t_0, \ldots, t_{m-1}\}$ are disjoint subsets of $\{0, \ldots, n-1\}$ and let $X, Y : \{0,1\}^n \to \mathbb{R}$ be random variables such that $X = F(x_{s_0}, \ldots, x_{s_{k-1}})$ and $Y = G(x_{t_0}, \ldots, x_{t_{m-1}})$ for some functions $F : \{0,1\}^k \to \mathbb{R}$ and $G : \{0,1\}^m \to \mathbb{R}$. Then X and Y are independent.

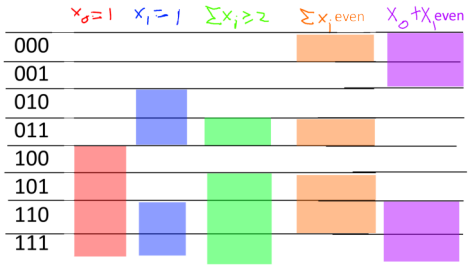


Figure 18.5: Consider the sample space $\{0,1\}^n$ and the events A, B, C, D, E corresponding to A: $x_0 = 1$, B: $x_1 = 1$, C: $x_0 + x_1 + x_2 \geq 2$, D: $x_0 + x_1 + x_2 = 0 \mod 2$ and E: $x_0 + x_1 = 0 \mod 2$. We can see that A and B are independent, C is positively correlated with A and positively correlated with B, the three events A, B, D are mutually independent, and while every pair out of A, B, E is independent, the three events A, B, E are not mutually independent since their intersection has probability $\frac{2}{8} = \frac{1}{4}$ instead of $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$.

P

The notation in the lemma's statement is a bit cumbersome, but at the end of the day, it simply says that if X and Y are random variables that depend on two disjoint sets S and T of coins (for example, X might be the sum of the first $n/2$ coins, and Y might be the largest consecutive stretch of zeroes in the second $n/2$ coins), then they are independent.

Proof of Lemma 18.6. Let $a, b \in \mathbb{R}$, and let $A = \{x \in \{0,1\}^k : F(x) = a\}$ and $B = \{x \in \{0,1\}^m : G(x) = b\}$. Since S and T are disjoint, we can reorder the indices so that $S = \{0, \ldots, k-1\}$ and $T = \{k, \ldots, k+m-1\}$ without affecting any of the probabilities. Hence we can write $\Pr[X = a \wedge Y = b] = |C|/2^n$ where $C = \{x_0, \ldots, x_{n-1} : (x_0, \ldots, x_{k-1}) \in A \wedge (x_k, \ldots, x_{k+m-1}) \in B\}$. Another way to write this using string concatenation is that $C = \{xyz : x \in A, y \in B, z \in \{0,1\}^{n-k-m}\}$, and hence $|C| = |A||B|2^{n-k-m}$, which means that

$$\frac{|C|}{2^n} = \frac{|A|}{2^k}\frac{|B|}{2^m}\frac{2^{n-k-m}}{2^{n-k-m}} = \Pr[X = a]\Pr[Y = b].$$

■

If X and Y are independent random variables then (letting S_X, S_Y denote the sets of all numbers that have positive probability of being the output of X and Y, respectively):

$$\mathbb{E}[XY] = \sum_{a \in S_X, b \in S_Y} \Pr[X = a \wedge Y = b] \cdot ab \stackrel{(1)}{=} \sum_{a \in S_X, b \in S_Y} \Pr[X = a]\Pr[Y = b] \cdot ab \stackrel{(2)}{=}$$
$$\left(\sum_{a \in S_X} \Pr[X = a] \cdot a\right)\left(\sum_{b \in S_Y} \Pr[Y = b] \cdot b\right) \stackrel{(3)}{=}$$
$$\mathbb{E}[X]\,\mathbb{E}[Y]$$

where the first equality ($=^{(1)}$) follows from the independence of X and Y, the second equality ($=^{(2)}$) follows by "opening the parentheses" of the right-hand side, and the third equality ($=^{(3)}$) follows from the definition of expectation. (This is not an "if and only if"; see Exercise 18.3.)

Another useful fact is that if X and Y are independent random variables, then so are $F(X)$ and $G(Y)$ for all functions $F, G : \mathbb{R} \to \mathbb{R}$. This is intuitively true since learning $F(X)$ can only provide us with less information than does learning X itself. Hence, if learning X does not teach us anything about Y (and so also about $G(Y)$) then neither will learning $F(X)$. Indeed, to prove this we can write for every $a, b \in \mathbb{R}$:

$$\Pr[F(X)=a \wedge G(Y)=b] = \sum_{x \text{ s.t.} F(x)=a, y \text{ s.t. } G(y)=b} \Pr[X=x \wedge Y=y] =$$
$$\sum_{x \text{ s.t.} F(x)=a, y \text{ s.t. } G(y)=b} \Pr[X=x]\Pr[Y=y] =$$
$$\left(\sum_{x \text{ s.t.} F(x)=a} \Pr[X=x]\right) \cdot \left(\sum_{y \text{ s.t.} G(y)=b} \Pr[Y=y]\right) =$$
$$\Pr[F(X)=a]\Pr[G(Y)=b].$$

18.2.2 Collections of independent random variables

We can extend the notions of independence to more than two random variables: we say that the random variables $X_0, \ldots, X_{n-1}$ are *mutually independent* if for every $a_0, \ldots, a_{n-1} \in \mathbb{R}$,

$$\Pr[X_0 = a_0 \wedge \cdots \wedge X_{n-1} = a_{n-1}] = \Pr[X_0 = a_0] \cdots \Pr[X_{n-1} = a_{n-1}].$$

And similarly, we have that

Lemma 18.7 — Expectation of product of independent random variables. If $X_0, \ldots, X_{n-1}$ are mutually independent then

$$\mathbb{E}[\prod_{i=0}^{n-1} X_i] = \prod_{i=0}^{n-1} \mathbb{E}[X_i].$$

Lemma 18.8 — Functions preserve independence. If $X_0, \ldots, X_{n-1}$ are mutually independent, and $Y_0, \ldots, Y_{n-1}$ are defined as $Y_i = F_i(X_i)$ for some functions $F_0, \ldots, F_{n-1} : \mathbb{R} \to \mathbb{R}$, then $Y_0, \ldots, Y_{n-1}$ are mutually independent as well.

P

We leave proving Lemma 18.7 and Lemma 18.8 as Exercise 18.6 and Exercise 18.7. It is a good idea for you stop now and do these exercises to make sure you are comfortable with the notion of independence, as we will use it heavily later on in this course.

18.3 CONCENTRATION AND TAIL BOUNDS

The name "expectation" is somewhat misleading. For example, suppose that you and I place a bet on the outcome of 10 coin tosses, where if they all come out to be 1's then I pay you 100,000 dollars and otherwise you pay me 10 dollars. If we let $X : \{0,1\}^{10} \to \mathbb{R}$ be the random variable denoting your gain, then we see that

$$\mathbb{E}[X] = 2^{-10} \cdot 100000 - (1 - 2^{-10})10 \sim 90.$$

But we don't really "expect" the result of this experiment to be for you to gain 90 dollars. Rather, 99.9% of the time you will pay me 10 dollars, and you will hit the jackpot 0.1% of the times.

However, if we repeat this experiment again and again (with fresh and hence *independent* coins), then in the long run we do expect your average earning to be close to 90 dollars, which is the reason why casinos can make money in a predictable way even though every individual bet is random. For example, if we toss n independent and unbiased coins, then as n grows, the number of coins that come up ones will be more and more *concentrated* around $n/2$ according to the famous "bell curve" (see Fig. 18.6).

Much of probability theory is concerned with so called *concentration* or *tail* bounds, which are upper bounds on the probability that a random variable X deviates too much from its expectation. The first and simplest one of them is Markov's inequality:

Theorem 18.9 — Markov's inequality. If X is a non-negative random variable then for every $k > 1$, $\Pr[X \geq k\,\mathbb{E}[X]] \leq 1/k$.

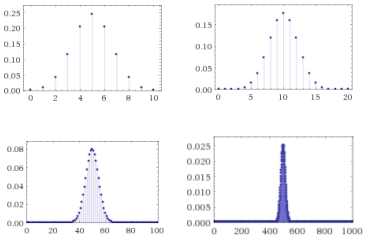


Figure 18.6: The probabilities that we obtain a particular sum when we toss $n = 10, 20, 100, 1000$ coins converge quickly to the Gaussian/normal distribution.

P

Markov's Inequality is actually a very natural statement (see also Fig. 18.7). For example, if you know that the average (not the median!) household income in the US is 70,000 dollars, then in particular you can deduce that at most 25 percent of households make more than 280,000 dollars, since otherwise, even if the remaining 75 percent had zero income, the top 25 percent alone would cause the average income to be larger than 70,000 dollars. From this example you can already see that in many situations, Markov's inequality will not be *tight* and the probability of deviating from expectation will be much smaller: see the Chebyshev and Chernoff inequalities below.

Proof of Theorem 18.9. Let $\mu = \mathbb{E}[X]$ and define $Y = 1_{X \geq k\mu}$. That is, $Y(x) = 1$ if $X(x) \geq k\mu$ and $Y(x) = 0$ otherwise. Note that by definition, for every x, $Y(x) \leq X/(k\mu)$. We need to show $\mathbb{E}[Y] \leq 1/k$. But this follows since $\mathbb{E}[Y] \leq \mathbb{E}[X/k(\mu)] = \mathbb{E}[X]/(k\mu) = \mu/(k\mu) = 1/k$. ■

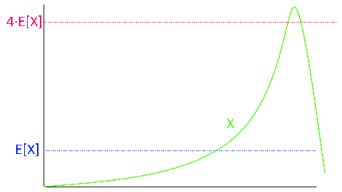


Figure 18.7: Markov's Inequality tells us that a non-negative random variable X cannot be much larger than its expectation, with high probability. For example, if the expectation of X is μ, then the probability that $X > 4\mu$ must be at most $1/4$, as otherwise just the contribution from this part of the sample space will be too large.

The averaging principle. While the expectation of a random variable X is hardly always the "typical value", we can show that X is guaranteed to achieve a value that is at least its expectation with positive probability. For example, if the average grade in an exam is 87 points, at least one student got a grade 87 or more on the exam. This is known

as the *averaging principle*, and despite its simplicity it is surprisingly useful.

Lemma 18.10 Let X be a random variable, then $\Pr[X \geq \mathbb{E}[X]] > 0$.

Proof. Suppose towards the sake of contradiction that $\Pr[X < \mathbb{E}[X]] = 1$. Then the random variable $Y = \mathbb{E}[X] - X$ is always positive. By linearity of expectation $\mathbb{E}[Y] = \mathbb{E}[X] - \mathbb{E}[X] = 0$. Yet by Markov, a non-negative random variable Y with $\mathbb{E}[Y] = 0$ must equal 0 with probability 1, since the probability that $Y > k \cdot 0 = 0$ is at most $1/k$ for every $k > 1$. Hence we get a contradiction to the assumption that Y is always positive. ■

18.3.1 Chebyshev's Inequality

Markov's inequality says that a (non-negative) random variable X can't go too crazy and be, say, a million times its expectation, with significant probability. But ideally we would like to say that with high probability, X should be very close to its expectation, e.g., in the range $[0.99\mu, 1.01\mu]$ where $\mu = \mathbb{E}[X]$. In such a case we say that X is *concentrated*, and hence its expectation (i.e., mean) will be close to its *median* and other ways of measuring X's "typical value". *Chebyshev's inequality* can be thought of as saying that X is concentrated if it has a small *standard deviation*.

A standard way to measure the deviation of a random variable from its expectation is by using its *standard deviation*. For a random variable X, we define the *variance* of X as $\mathrm{Var}[X] = \mathbb{E}[(X - \mu)^2]$ where $\mu = \mathbb{E}[X]$; i.e., the variance is the average squared distance of X from its expectation. The *standard deviation* of X is defined as $\sigma[X] = \sqrt{\mathrm{Var}[X]}$. (This is well-defined since the variance, being an average of a square, is always a non-negative number.)

Using Chebyshev's inequality, we can control the probability that a random variable is too many standard deviations away from its expectation.

Theorem 18.11 — Chebyshev's inequality. Suppose that $\mu = \mathbb{E}[X]$ and $\sigma^2 = \mathrm{Var}[X]$. Then for every $k > 0$, $\Pr[|X - \mu| \geq k\sigma] \leq 1/k^2$.

Proof. The proof follows from Markov's inequality. We define the random variable $Y = (X - \mu)^2$. Then $\mathbb{E}[Y] = \mathrm{Var}[X] = \sigma^2$, and hence by Markov the probability that $Y > k^2\sigma^2$ is at most $1/k^2$. But clearly $(X - \mu)^2 \geq k^2\sigma^2$ if and only if $|X - \mu| \geq k\sigma$. ■

One example of how to use Chebyshev's inequality is the setting when $X = X_1 + \cdots + X_n$ where X_i's are *independent and identically*

distributed (i.i.d for short) variables with values in $[0,1]$ where each has expectation $1/2$. Since $\mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = n/2$, we would like to say that X is very likely to be in, say, the interval $[0.499n, 0.501n]$. Using Markov's inequality directly will not help us, since it will only tell us that X is very likely to be at most $100n$ (which we already knew, since it always lies between 0 and n). However, since $X_1, \ldots, X_n$ are independent,

$$\mathrm{Var}[X_1 + \cdots + X_n] = \mathrm{Var}[X_1] + \cdots + \mathrm{Var}[X_n] \,. \tag{18.1}$$

(We leave showing this to the reader as Exercise 18.8.)

For every random variable X_i in $[0,1]$, $\mathrm{Var}[X_i] \leq 1$ (if the variable is always in $[0,1]$, it can't be more than 1 away from its expectation), and hence (18.1) implies that $\mathrm{Var}[X] \leq n$ and hence $\sigma[X] \leq \sqrt{n}$. For large n, $\sqrt{n} \ll 0.001n$, and in particular if $\sqrt{n} \leq 0.001n/k$, we can use Chebyshev's inequality to bound the probability that X is not in $[0.499n, 0.501n]$ by $1/k^2$.

18.3.2 The Chernoff bound

Chebyshev's inequality already shows a connection between independence and concentration, but in many cases we can hope for a quantitatively much stronger result. If, as in the example above, $X = X_1 + \ldots + X_n$ where the X_i's are bounded i.i.d random variables of mean $1/2$, then as n grows, the distribution of X would be roughly the *normal* or *Gaussian* distribution— that is, distributed according to the *bell curve* (see Fig. 18.6 and Fig. 18.8). This distribution has the property of being *very* concentrated in the sense that the probability of deviating k standard deviations from the mean is not merely $1/k^2$ as is guaranteed by Chebyshev, but rather is roughly e^{-k^2}. Specifically, for a normal random variable X of expectation μ and standard deviation σ, the probability that $|X - \mu| \geq k\sigma$ is at most $2e^{-k^2/2}$. That is, we have an *exponential decay* of the probability of deviation.

The following extremely useful theorem shows that such exponential decay occurs every time we have a sum of independent and bounded variables. This theorem is known under many names in different communities, though it is mostly called the Chernoff bound in the computer science literature:

Theorem 18.12 — Chernoff/Hoeffding bound. If $X_0, \ldots, X_{n-1}$ are i.i.d random variables such that $X_i \in [0,1]$ and $\mathbb{E}[X_i] = p$ for every i, then for every $\epsilon > 0$

$$\Pr\left[\left|\sum_{i=0}^{n-1} X_i - pn\right| > \epsilon n\right] \leq 2 \cdot e^{-2\epsilon^2 n}. \tag{18.2}$$

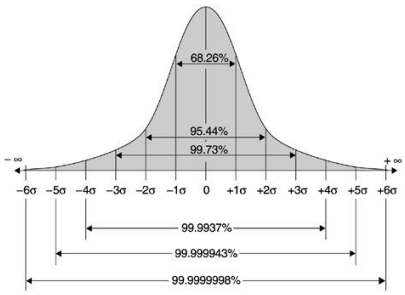


Figure 18.8: In the *normal distribution* or the bell curve, the probability of deviating k standard deviations from the expectation shrinks *exponentially* in k^2, and specifically with probability at least $1 - 2e^{-k^2/2}$, a random variable X of expectation μ and standard deviation σ satisfies $\mu - k\sigma \leq X \leq \mu + k\sigma$. This figure gives more precise bounds for $k = 1, 2, 3, 4, 5, 6$. (Image credit:Imran Baghirov)

We omit the proof, which appears in many texts, and uses Markov's inequality on i.i.d random variables $Y_0, \ldots, Y_n$ that are of the form $Y_i = e^{\lambda X_i}$ for some carefully chosen parameter λ. See Exercise 18.11 for a proof of the simple (but highly useful and representative) case where each X_i is $\{0,1\}$ valued and $p = 1/2$. (See also Exercise 18.12 for a generalization.)

R

Remark 18.13 — Slight simplification of Chernoff. Since e is roughly 2.7 (and in particular larger than 2), (18.2) would still be true if we replaced its right-hand side with $e^{-2\epsilon^2 n+1}$. For $n > 1/\epsilon^2$, the equation will still be true if we replaced the right-hand side with the simpler $e^{-\epsilon^2 n}$. Hence we will sometimes use the Chernoff bound as stating that for $X_0, \ldots, X_{n-1}$ and p as above, $n > 1/\epsilon^2$ then

$$\Pr\left[\left|\sum_{i=0}^{n-1} X_i - pn\right| > \epsilon n\right] \leq e^{-\epsilon^2 n}. \tag{18.3}$$

18.3.3 Application: Supervised learning and empirical risk minimization

Here is a nice application of the Chernoff bound. Consider the task of *supervised learning*. You are given a set S of n samples of the form $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ drawn from some unknown distribution D over pairs (x, y). For simplicity we will assume that $x_i \in \{0,1\}^m$ and $y_i \in \{0,1\}$. (We use here the concept of general distribution over the finite set $\{0,1\}^{m+1}$ as discussed in Section 18.1.3.) The goal is to find a *classifier* $h : \{0,1\}^m \to \{0,1\}$ that will minimize the *test error* which is the probability $L(h)$ that $h(x) \neq y$ where (x, y) is drawn from the distribution D. That is, $L(h) = \Pr_{(x,y)\sim D}[h(x) \neq y]$.

One way to find such a classifier is to consider a *collection* $\mathcal{C}$ of potential classifiers and look at the classifier h in $\mathcal{C}$ that does best on the *training set* S. The classifier h is known as the *empirical risk minimizer* (see also Section 12.1.6) . The Chernoff bound can be used to show that as long as the number n of samples is sufficiently larger than the logarithm of $|\mathcal{C}|$, the test error $L(h)$ will be close to its *training error* $\hat{L}_S(h)$, which is defined as the fraction of pairs $(x_i, y_i) \in S$ that it fails to classify. (Equivalently, $\hat{L}_S(h) = \frac{1}{n}\sum_{i\in[n]} |h(x_i) - y_i|$.)

Theorem 18.14 — Generalization of ERM. Let D be any distribution over pairs $(x, y) \in \{0,1\}^{m+1}$ and $\mathcal{C}$ be any set of functions mapping $\{0,1\}^m$ to $\{0,1\}$. Then for every $\epsilon, \delta > 0$, if $n > \frac{\log|\mathcal{C}|\log(1/\delta)}{\epsilon^2}$ and S is a set of $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ samples that are drawn indepen-

dently from D then

$$\Pr_S \left[\forall_{h \in \mathcal{C}} |L(h) - \hat{L}_S(h)| \leq \epsilon \right] > 1 - \delta \ ,$$

where the probability is taken over the choice of the set of samples S.

In particular if $|\mathcal{C}| \leq 2^k$ and $n > \frac{k \log(1/\delta)}{\epsilon^2}$ then with probability at least $1-\delta$, the classifier $h_* \in \mathcal{C}$ that minimizes that empirical test error $\hat{L}_S(C)$ satisfies $L(h_*) \leq \hat{L}_S(h_*) + \epsilon$, and hence its test error is at most ϵ worse than its training error.

Proof Idea:

The idea is to combine the Chernoff bound with the union bound. Let $k = \log |\mathcal{C}|$. We first use the Chernoff bound to show that for every *fixed* $h \in \mathcal{C}$, if we choose S at random then the probability that $|L(h) - \hat{L}_S(h)| > \epsilon$ will be smaller than $\frac{\delta}{2^k}$. We can then use the union bound over all the 2^k members of $\mathcal{C}$ to show that this will be the case *for every* h.

⋆

Proof of Theorem 18.14. Set $k = \log |\mathcal{C}|$ and so $n > k \log(1/\delta)/\epsilon^2$. We start by making the following claim.

CLAIM: For every $h \in \mathcal{C}$, the probability over S that $|L(h) - \hat{L}_S(h)| \geq \epsilon$ is smaller than $\delta/2^k$.

We prove the claim using the Chernoff bound. Specifically, for every such h, let us define a collection of random variables $X_0, \ldots, X_{n-1}$ as follows:

$$X_i = \begin{cases} 1 & h(x_i) \neq y_i \\ 0 & \text{otherwise} \end{cases} .$$

Since the samples $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ are drawn independently from the same distribution D, the random variables $X_0, \ldots, X_{n-1}$ are independently and identically distributed. Moreover, for every i, $\mathbb{E}[X_i] = L(h)$. Hence by the Chernoff bound (see (18.3)), the probability that $|\sum_{i=0}^{n-1} X_i - n \cdot L(h)| \geq \epsilon n$ is at most $e^{-\epsilon^2 n} < e^{-k \log(1/\delta)} < \delta/2^k$ (using the fact that $e > 2$). Since $\hat{L}(h) = \frac{1}{n} \sum_{i \in [n]} X_i$, this completes the proof of the claim.

Given the claim, the theorem follows from the union bound. Indeed, for every $h \in \mathcal{C}$, define the "bad event" B_h to be the event (over the choice of S) that $|L(h) - \hat{L}_S(h)| > \epsilon$. By the claim $\Pr[B_h] < \delta/2^k$, and hence by the union bound the probability that the *union* of B_h for all $h \in \mathcal{H}$ happens is smaller than $|\mathcal{C}|\delta/2^k = \delta$. If for every $h \in \mathcal{C}$, B_h does *not* happen, it means that for every $h \in \mathcal{H}$, $|L(h) - \hat{L}_S(h)| \leq \epsilon$,

and so the probability of the latter event is larger than $1 - \delta$ which is what we wanted to prove.

■

Chapter Recap

- A basic probabilistic experiment corresponds to tossing n coins or choosing x uniformly at random from $\{0,1\}^n$.
- *Random variables* assign a real number to every result of a coin toss. The *expectation* of a random variable X is its average value.
- There are several *concentration* results, also known as *tail bounds* showing that under certain conditions, random variables deviate significantly from their expectation only with small probability.

18.4 EXERCISES

Exercise 18.1 Suppose that we toss three independent fair coins $a, b, c \in \{0,1\}$. What is the probability that the XOR of a,b, and c is equal to 1? What is the probability that the AND of these three values is equal to 1? Are these two events independent?

■

Exercise 18.2 Give an example of random variables $X, Y : \{0,1\}^3 \to \mathbb{R}$ such that $\mathbb{E}[XY] \neq \mathbb{E}[X]\,\mathbb{E}[Y]$.

■

Exercise 18.3 Give an example of random variables $X, Y : \{0,1\}^3 \to \mathbb{R}$ such that X and Y are *not* independent but $\mathbb{E}[XY] = \mathbb{E}[X]\,\mathbb{E}[Y]$.

■

Exercise 18.4 Let n be an odd number, and let $X : \{0,1\}^n \to \mathbb{R}$ be the random variable defined as follows: for every $x \in \{0,1\}^n$, $X(x) = 1$ if $\sum_{i=0} x_i > n/2$ and $X(x) = 0$ otherwise. Prove that $\mathbb{E}[X] = 1/2$.

■

Exercise 18.5 — standard deviation. 1. Give an example for a random variable X such that X's standard deviation is *equal to* $\mathbb{E}[|X - \mathbb{E}[X]|]$

2. Give an example for a random variable X such that X's standard deviation is *not equal to* $\mathbb{E}[|X - \mathbb{E}[X]|]$

■

Exercise 18.6 — Product of expectations. Prove Lemma 18.7

■

Exercise 18.7 — Transformations preserve independence. Prove Lemma 18.8

■

Exercise 18.8 — Variance of independent random variables. Prove that if $X_0, \ldots, X_{n-1}$ are independent random variables then $\mathrm{Var}[X_0 + \cdots + X_{n-1}] = \sum_{i=0}^{n-1} \mathrm{Var}[X_i]$.

Exercise 18.9 — Entropy (challenge). Recall the definition of a distribution μ over some finite set S. Shannon defined the *entropy* of a distribution μ, denoted by $H(\mu)$, to be $\sum_{x \in S} \mu(x) \log(1/\mu(x))$. The idea is that if μ is a distribution of entropy k, then encoding members of μ will require k bits, in an amortized sense. In this exercise we justify this definition. Let μ be such that $H(\mu) = k$.

1. Prove that for every one to one function $F : S \to \{0,1\}^*$, $\mathbb{E}_{x \sim \mu} |F(x)| \geq k$.
2. Prove that for every ϵ, there is some n and a one-to-one function $F : S^n \to \{0,1\}^*$, such that $\mathbb{E}_{x \sim \mu^n} |F(x)| \leq n(k+\epsilon)$, where $x \sim \mu$ denotes the experiments of choosing $x_0, \ldots, x_{n-1}$ each independently from S using the distribution μ.

Exercise 18.10 — Entropy approximation to binomial. Let $H(p) = p \log(1/p) + (1-p)\log(1/(1-p))$.[1] Prove that for every $p \in (0,1)$ and $\epsilon > 0$, if n is large enough then[2]

$$2^{(H(p)-\epsilon)n} \binom{n}{pn} \leq 2^{(H(p)+\epsilon)n}$$

where $\binom{n}{k}$ is the binomial coefficient $\frac{n!}{k!(n-k)!}$ which is equal to the number of k-size subsets of $\{0, \ldots, n-1\}$.

[1] While you don't need this to solve this exercise, this is the function that maps p to the entropy (as defined in Exercise 18.9) of the p-biased coin distribution over $\{0,1\}$, which is the function $\mu : \{0,1\} \to [0,1]$ s.y. $\mu(0) = 1-p$ and $\mu(1) = p$.

[2] **Hint:** Use Stirling's formula for approximating the factorial function.

Exercise 18.11 — Chernoff using Stirling.

1. Prove that $\Pr_{x \sim \{0,1\}^n}[\sum x_i = k] = \binom{n}{k} 2^{-n}$.
2. Use this and Exercise 18.10 to prove (an approximate version of) the Chernoff bound for the case that $X_0, \ldots, X_{n-1}$ are i.i.d. random variables over $\{0,1\}$ each equaling 0 and 1 with probability $1/2$. That is, prove that for every $\epsilon > 0$, and $X_0, \ldots, X_{n-1}$ as above, $\Pr[|\sum_{i=0}^{n-1} X_i - \frac{n}{2}| > \epsilon n] < 2^{0.1 \cdot \epsilon^2 n}$.

Exercise 18.12 — Poor man's Chernoff. Exercise 18.11 establishes the Chernoff bound for the case that $X_0, \ldots, X_{n-1}$ are i.i.d variables over $\{0,1\}$ with expectation $1/2$. In this exercise we use a slightly different method (bounding the *moments* of the random variables) to establish a version of Chernoff where the random variables range over $[0,1]$ and their expectation is some number $p \in [0,1]$ that may be different than $1/2$. Let $X_0, \ldots, X_{n-1}$ be i.i.d random variables with $\mathbb{E} X_i = p$ and $\Pr[0 \leq X_i \leq 1] = 1$. Define $Y_i = X_i - p$.

1. Prove that for every $j_0, \ldots, j_{n-1} \in \mathbb{N}$, if there exists one i such that j_i is odd then $\mathbb{E}[\prod_{i=0}^{n-1} Y_i^{j_i}] = 0$.

2. Prove that for every k, $\mathbb{E}[(\sum_{i=0}^{n-1} Y_i)^k] \leq (10kn)^{k/2}$.[3]

3. Prove that for every $\epsilon > 0$, $\Pr[|\sum_i Y_i| \geq \epsilon n] \geq 2^{-\epsilon^2 n/(10000 \log 1/\epsilon)}$.[4]

[3] **Hint:** Bound the number of tuples $j_0, \ldots, j_{n-1}$ such that every j_i is even and $\sum j_i = k$ using the Binomial coefficient and the fact that in any such tuple there are at most $k/2$ distinct indices.

[4] **Hint:** Set $k = 2\lceil \epsilon^2 n/1000 \rceil$ and then show that if the event $|\sum Y_i| \geq \epsilon n$ happens then the random variable $(\sum Y_i)^k$ is a factor of ϵ^{-k} larger than its expectation.

■

Exercise 18.13 — Sampling. Suppose that a country has 300,000,000 citizens, 52 percent of which prefer the color "green" and 48 percent of which prefer the color "orange". Suppose we sample n random citizens and ask them their favorite color (assume they will answer truthfully). What is the smallest value n among the following choices so that the probability that the majority of the sample answers "green" is at most 0.05?

a. 1,000

b. 10,000

c. 100,000

d. 1,000,000

■

Exercise 18.14 Would the answer to Exercise 18.13 change if the country had 300,000,000,000 citizens?

■

Exercise 18.15 — Sampling (2). Under the same assumptions as Exercise 18.13, what is the smallest value n among the following choices so that the probability that the majority of the sample answers "green" is at most 2^{-100}?

a. 1,000

b. 10,000

c. 100,000

d. 1,000,000

e. It is impossible to get such low probability since there are fewer than 2^{100} citizens.

■

18.5 BIBLIOGRAPHICAL NOTES

There are many sources for more information on discrete probability, including the texts referenced in Section 1.9. One particularly recommended source for probability is Harvard's STAT 110 class, whose lectures are available on youtube and whose book is available online.

The version of the Chernoff bound that we stated in Theorem 18.12 is sometimes known as Hoeffding's Inequality. Other variants of the Chernoff bound are known as well, but all of them are equally good for the applications of this book.

Learning Objectives:

- See examples of randomized algorithms
- Get more comfort with analyzing probabilistic processes and tail bounds
- Success amplification using tail bounds

19
Probabilistic computation

> *"in 1946 .. (I asked myself) what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method ... might not be to lay it out say one hundred times and simply observe and count"*, Stanislaw Ulam, 1983

> *"The salient features of our method are that it is probabilistic ... and with a controllable miniscule probability of error."*, Michael Rabin, 1977

In early computer systems, much effort was taken to drive *out* randomness and noise. Hardware components were prone to non-deterministic behavior from a number of causes, whether it was vacuum tubes overheating or actual physical bugs causing short circuits (see Fig. 19.1). This motivated John von Neumann, one of the early computing pioneers, to write a paper on how to *error correct* computation, introducing the notion of *redundancy*.

So it is quite surprising that randomness turned out not just a hindrance but also a *resource* for computation, enabling us to achieve tasks much more efficiently than previously known. One of the first applications involved the very same John von Neumann. While he was sick in bed and playing cards, Stan Ulam came up with the observation that calculating statistics of a system could be done much faster by running several randomized simulations. He mentioned this idea to von Neumann, who became very excited about it; indeed, it turned out to be crucial for the neutron transport calculations that were needed for development of the Atom bomb and later on the hydrogen bomb. Because this project was highly classified, Ulam, von Neumann and their collaborators came up with the codeword "Monte Carlo" for this approach (based on the famous casinos where Ulam's uncle gambled). The name stuck, and randomized algorithms are known as Monte Carlo algorithms to this day.[1]

In this chapter, we will see some examples of randomized algorithms that use randomness to compute a quantity in a faster or simpler way than was known otherwise. We will describe the algorithms

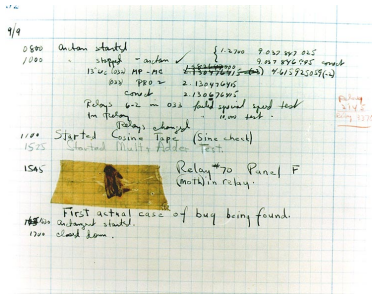


Figure 19.1: A 1947 entry in the log book of the Harvard MARK II computer containing an actual bug that caused a hardware malfunction. By Courtesy of the Naval Surface Warfare Center.

[1] Some texts also talk about "Las Vegas algorithms" that always return the right answer but whose running time is only polynomial on the average. Since this Monte Carlo vs Las Vegas terminology is confusing, we will not use these terms anymore, and simply talk about randomized algorithms.

in an informal / "pseudo-code" way, rather than as Turing macines or NAND-TM/NAND-RAM programs. In Chapter 20 we will discuss how to augment the computational models we saw before to incorporate the ability to "toss coins".

This chapter: A non-mathy overview

This chapter gives some examples of randomized algorithms to get a sense of why probability can be useful for computation. We will also see the technique of *success amplification* which is key for many randomized algorithms.

19.1 FINDING APPROXIMATELY GOOD MAXIMUM CUTS

We start with the following example. Recall the *maximum cut problem* of finding, given a graph $G = (V, E)$, the cut that maximizes the number of edges. This problem is **NP**-hard, which means that we do not know of any efficient algorithm that can solve it, but randomization enables a simple algorithm that can cut at least half of the edges:

Theorem 19.1 — Approximating max cut. There is an efficient probabilistic algorithm that on input an n-vertex m-edge graph G, outputs a cut $(S, \overline{S})$ that cuts at least $m/2$ of the edges of G in expectation.

Proof Idea:

We simply choose a *random cut*: we choose a subset S of vertices by choosing every vertex v to be a member of S with probability $1/2$ independently. It's not hard to see that each edge is cut with probability $1/2$ and so the expected number of cut edges is $m/2$.

⋆

Proof of Theorem 19.1. The algorithm is extremely simple:

Algorithm Random Cut:

Input: Graph $G = (V, E)$ with n vertices and m edges. Denote $V = \{v_0, v_1, \ldots, v_{n-1}\}$.

Operation:

1. Pick x uniformly at random in $\{0,1\}^n$.
2. Let $S \subseteq V$ be the set $\{v_i : x_i = 1 , i \in [n]\}$ that includes all vertices corresponding to coordinates of x where $x_i = 1$.
3. Output the cut $(S, \overline{S})$.

We claim that the expected number of edges cut by the algorithm is $m/2$. Indeed, for every edge $e \in E$, let X_e be the random variable such that $X_e(x) = 1$ if the edge e is cut by x, and $X_e(x) = 0$ otherwise. For

every such edge $e = \{i, j\}$, $X_e(x) = 1$ if and only if $x_i \neq x_j$. Since the pair (x_i, x_j) obtains each of the values $00, 01, 10, 11$ with probability $1/4$, the probability that $x_i \neq x_j$ is $1/2$ and hence $\mathbb{E}[X_e] = 1/2$. If we let X be the random variable corresponding to the total number of edges cut by S, then $X = \sum_{e \in E} X_e$ and hence by linearity of expectation

$$\mathbb{E}[X] = \sum_{e \in E} \mathbb{E}[X_e] = m(1/2) = m/2 \ .$$

■

Randomized algorithms work in the worst case. It is tempting to think of a randomized algorithm such as the one of Theorem 19.1 as an algorithm that works for a "random input graph" but it is actually much better than that. The expectation in this theorem is *not* taken over the choice of the graph, but rather only over the *random choices of the algorithm*. In particular, *for every graph* G, the algorithm is guaranteed to cut half of the edges of the input graph in expectation. That is,

Big Idea 24 A randomized algorithm outputs the correct value with good probability on *every possible input*.

We will define more formally what "good probability" means in Chapter 20 but the crucial point is that this probability is always only taken over the random choices of the algorithm, while the input is *not* chosen at random.

19.1.1 Amplifying the success of randomized algorithms

Theorem 19.1 gives us an algorithm that cuts $m/2$ edges in *expectation*. But, as we saw before, expectation does not immediately imply concentration, and so a priori, it may be the case that when we run the algorithm, most of the time we don't get a cut matching the expectation. Luckily, we can *amplify* the probability of success by repeating the process several times and outputting the best cut we find. We start by arguing that the probability the algorithm above succeeds in cutting at least $m/2$ edges is not *too* tiny.

Lemma 19.2 The probability that a random cut in an m edge graph cuts at least $m/2$ edges is at least $1/(2m)$.

Proof Idea:

To see the idea behind the proof, think of the case that $m = 1000$. In this case one can show that we will cut at least 500 edges with probability at least 0.001 (and so in particular larger than $1/(2m) = 1/2000$). Specifically, if we assume otherwise, then this means that with probability more than 0.999 the algorithm cuts 499 or fewer edges. But since

we can never cut more than the total of 1000 edges, given this assumption, the highest value of the expected number of edges cut is if we cut exactly 499 edges with probability 0.999 and cut 1000 edges with probability 0.001. Yet even in this case the expected number of edges will be $0.999 \cdot 499 + 0.001 \cdot 1000 < 500$, which contradicts the fact that we've calculated the expectation to be at least 500 in Theorem 19.1.

⋆

Proof of Lemma 19.2. Let p be the probability that we cut at least $m/2$ edges and suppose, towards a contradiction, that $p < 1/(2m)$. Since the number of edges cut is an integer, and $m/2$ is a multiple of 0.5, by definition of p, with probability $1-p$ we cut at most $m/2 - 0.5$ edges. Moreover, since we can never cut more than m edges, under our assumption that $p < 1/(2m)$, we can bound the expected number of edges cut by

$$pm + (1-p)(m/2 - 0.5) \leq pm + m/2 - 0.5$$

But if $p < 1/(2m)$ then $pm < 0.5$ and so the right-hand side is smaller than $m/2$, which contradicts the fact that (as proven in Theorem 19.1) the expected number of edges cut is at least $m/2$.

■

19.1.2 Success amplification

Lemma 19.2 shows that our algorithm succeeds at least *some* of the time, but we'd like to succeed almost *all* of the time. The approach to do that is to simply *repeat* our algorithm many times, with fresh randomness each time, and output the best cut we get in one of these repetitions. It turns out that with extremely high probability we will get a cut of size at least $m/2$. For example, if we repeat this experiment $2000m$ times, then (using the inequality $(1-1/k)^k \leq 1/e \leq 1/2$) we can show that the probability that we will never cut at least $m/2$ edges, where $k = 2m$, is at most

$$(1 - 1/(2m))^{2000m} = (1-1/k)^{1000k} = ((1-1/k)^k)^{1000} \leq 2^{-1000} .$$

More generally, the same calculations can be used to show the following lemma:

Lemma 19.3 There is an algorithm that on input a graph $G = (V, E)$ and a number k, runs in time polynomial in $|V|$ and k and outputs a cut $(S, \overline{S})$ such that

$$\Pr[\text{number of edges cut by } (S, \overline{S}) \geq |E|/2] \geq 1 - 2^{-k} .$$

Proof of Lemma 19.3. The algorithm will work as follows:

Algorithm AMPLIFY RANDOM CUT:

Input: Graph $G = (V, E)$ with n vertices and m edges. Denote $V = \{v_0, v_1, \ldots, v_{n-1}\}$. Number $k > 0$.

Operation:

1. Repeat the following $200km$ times:
   a. Pick x uniformly at random in $\{0,1\}^n$.
   b. Let $S \subseteq V$ be the set $\{v_i \ : \ x_i = 1 \ , \ i \in [n]\}$ that includes all vertices corresponding to coordinates of x where $x_i = 1$.
   c. If $(S, \overline{S})$ cuts at least $m/2$ then halt and output $(S, \overline{S})$.
2. Output "failed"

We leave completing the analysis as an exercise to the reader (see Exercise 19.1).

■

19.1.3 Two-sided amplification

The analysis above relied on the fact that the maximum cut has *one sided error*. By this we mean that if we get a cut of size at least $m/2$ then we know we have succeeded. This is common for randomized algorithms, but is not the only case. In particular, consider the task of computing some Boolean function $F : \{0,1\}^* \to \{0,1\}$. A randomized algorithm A for computing F, given input x, might toss coins and succeed in outputting $F(x)$ with probability, say, 0.9. We say that A has *two sided errors* if there is positive probability that $A(x)$ outputs 1 when $F(x) = 0$, and positive probability that $A(x)$ outputs 0 when $F(x) = 1$. In such a case, to simplify A's success, we cannot simply repeat it k times and output 1 if a single one of those repetitions resulted in 1, nor can we output 0 if a single one of the repetitions resulted in 0. But we can output the *majority value* of these repetitions. By the Chernoff bound (Theorem 18.12), with probability *exponentially close* to 1 (i.e., $1 - 2^{\Omega(k)}$), the fraction of the repetitions where A will output $F(x)$ will be at least, say 0.89, and in such cases we will of course output the correct answer.

The above translates into the following theorem

Theorem 19.4 — Two-sided amplification. If $F : \{0,1\}^* \to \{0,1\}$ is a function such that there is a polynomial-time algorithm A satisfying

$$\Pr[A(x) = F(x)] \geq 0.51$$

for every $x \in \{0,1\}^*$, then there is a polynomial time algorithm B satisfying

$$\Pr[B(x) = F(x)] \geq 1 - 2^{-|x|}$$

for every $x \in \{0,1\}^*$.

We omit the proof of Theorem 19.4, since we will prove a more general result later on in Theorem 20.5.

19.1.4 What does this mean?

We have shown a probabilistic algorithm that on any m edge graph G, will output a cut of at least $m/2$ edges with probability at least $1 - 2^{-1000}$. Does it mean that we can consider this problem as "easy"? Should we be somewhat wary of using a probabilistic algorithm, since it can sometimes fail?

First of all, it is important to emphasize that this is still a *worst case* guarantee. That is, we are not assuming anything about the *input graph*: the probability is only due to the *internal randomness of the algorithm*. While a probabilistic algorithm might not seem as nice as a deterministic algorithm that is *guaranteed* to give an output, to get a sense of what a failure probability of 2^{-1000} means, note that:

- The chance of winning the Massachusetts Mega Millions lottery is one over $(75)^5 \cdot 15$, which is roughly 2^{-35}. So 2^{-1000} corresponds to winning the lottery about 300 times in a row, at which point you might not care so much about your algorithm failing.
- The chance for a U.S. resident to be struck by lightning is about $1/700000$, which corresponds to about a 2^{-45} chance that you'll be struck by lightning the very second that you're reading this sentence (after which again you might not care so much about the algorithm's performance).
- Since the earth is about 5 billion years old, we can estimate the chance that an asteroid of the magnitude that caused the dinosaurs' extinction will hit us this very second to be about 2^{-60}. It is quite likely that even a deterministic algorithm will fail if this happens.

So, in practical terms, a probabilistic algorithm is just as good as a deterministic one. But it is still a theoretically fascinating question whether randomized algorithms actually yield more power, or whether is it the case that for any computational problem that can be solved by a probabilistic algorithm, there is a deterministic algorithm with nearly the same performance.[2] For example, we will see in Exercise 19.2 that there is in fact a deterministic algorithm that can cut at least $m/2$ edges in an m-edge graph. We will discuss this question in generality in Chapter 20. For now, let us see a couple of examples

[2] This question does have some significance to practice, since hardware that generates high quality randomness at speed is non-trivial to construct.

where randomization leads to algorithms that are better in some sense than the known deterministic algorithms.

19.2 SOLVING SAT THROUGH RANDOMIZATION

The 3SAT problem is **NP** hard, and so it is unlikely that it has a polynomial (or even subexponential) time algorithm. But this does not mean that we can't do at least somewhat better than the trivial 2^n algorithm for n-variable 3SAT. The best known worst-case algorithms for 3SAT are randomized, and are related to the following simple algorithm, variants of which are also used in practice:

Algorithm WalkSAT:

Input: An n variable 3CNF formula φ.

Parameters: $T, S \in \mathbb{N}$

Operation:

1. Repeat the following T steps:
   a. Choose a random assignment $x \in \{0,1\}^n$ and repeat the following for S steps:
      1. If x satisfies φ then output x.
      2. Otherwise, choose a random clause $(\ell_i \vee \ell_j \vee \ell_k)$ that x does not satisfy, choose a random literal in ℓ_i, ℓ_j, ℓ_k and modify x to satisfy this literal.
2. If all the $T \cdot S$ repetitions above did not result in a satisfying assignment then output `Unsatisfiable`

The running time of this algorithm is $S \cdot T \cdot poly(n)$, and so the key question is how small we can make S and T so that the probability that WalkSAT outputs `Unsatisfiable` on a satisfiable formula φ is small. It is known that we can do so with $ST = \tilde{O}((4/3)^n) = \tilde{O}(1.333\ldots^n)$ (see Exercise 19.4 for a weaker result), but we'll show below a simpler analysis yielding $ST = \tilde{O}(\sqrt{3}^n) = \tilde{O}(1.74^n)$, which is still much better than the trivial 2^n bound.[3]

[3] At the time of this writing, the best known randomized algorithms for 3SAT run in time roughly $O(1.308^n)$, and the best known deterministic algorithms run in time $O(1.3303^n)$ in the worst case.

Theorem 19.5 — WalkSAT simple analysis. If we set $T = 100 \cdot \sqrt{3}^n$ and $S = n/2$, then the probability we output `Unsatisfiable` for a satisfiable φ is at most $1/2$.

Proof. Suppose that φ is a satisfiable formula and let x^* be a satisfying assignment for it. For every $x \in \{0,1\}^n$, denote by $\Delta(x, x^*)$ the number of coordinates that differ between x and x^*. The heart of the proof is the following claim:

Claim I: For every x, x^* as above, in every local improvement step, the value $\Delta(x, x^*)$ is decreased by one with probability at least $1/3$.

Proof of Claim I: Since x^* is a *satisfying* assignment, if C is a clause that x does *not* satisfy, then at least one of the variables involved in C must get different values in x and x^*. Thus when we change x by one of the three literals in the clause, we have probability at least $1/3$ of decreasing the distance.

The second claim is that our starting point is not that bad:

Claim 2: With probability at least $1/2$ over a random $x \in \{0,1\}^n$, $\Delta(x, x^*) \leq n/2$.

Proof of Claim II: Consider the map $FLIP : \{0,1\}^n \to \{0,1\}^n$ that simply "flips" all the bits of its input from 0 to 1 and vice versa. That is, $FLIP(x_0, \ldots, x_{n-1}) = (1 - x_0, \ldots, 1 - x_{n-1})$. Clearly $FLIP$ is one to one. Moreover, if x is of distance k to x^*, then $FLIP(x)$ is distance $n - k$ to x^*. Now let B be the "bad event" in which x is of distance $> n/2$ from x^*. Then the set $A = FLIP(B) = \{FLIP(x) \ : \ x \in B\}$ satisfies $|A| = |B|$ and that if $x \in A$ then x is of distance $< n/2$ from x^*. Since A and B are disjoint events, $\Pr[A] + \Pr[B] \leq 1$. Since they have the same cardinality, they have the same probability and so we get that $2\Pr[B] \leq 1$ or $\Pr[B] \leq 1/2$. (See also Fig. 19.2).

Claims I and II imply that each of the T iterations of the outer loop succeeds with probability at least $1/2 \cdot \sqrt{3}^{-n}$. Indeed, by Claim II, the original guess x will satisfy $\Delta(x, x^*) \leq n/2$ with probability $\Pr[\Delta(x, x^*) \leq n/2] \geq 1/2$. By Claim I, even conditioned on all the history so far, for each of the $S = n/2$ steps of the inner loop we have probability at least $\geq 1/3$ of being "lucky" and decreasing the distance (i.e. the output of Δ) by one. The chance we will be lucky in all $n/2$ steps is hence at least $(1/3)^{n/2} = \sqrt{3}^{-n}$.

Since any single iteration of the outer loop succeeds with probability at least $\frac{1}{2} \cdot \sqrt{3}^{-n}$, the probability that we never do so in $T = 100\sqrt{3}^n$ repetitions is at most $(1 - \frac{1}{2\sqrt{3}^n})^{100 \cdot \sqrt{3}^n} \leq (1/e)^{50}$.

■

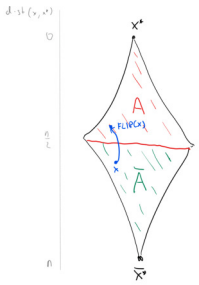


Figure 19.2: For every $x^* \in \{0,1\}^n$, we can sort all strings in $\{0,1\}^n$ according to their distance from x^* (top to bottom in the above figure), where we let $A = \{x \in \{0,1\}^n \mid dist(x, x^* \leq n/2\}$ be the "top half" of strings. If we define $FLIP : \{0,1\}^n \to \{0,1\}$ to be the map that "flips" the bits of a given string x then it maps every $x \in \overline{A}$ to an output $FLIP(x) \in A$ in a one-to-one way, and so it demonstrates that $|\overline{A}| \leq |A|$ which implies that $\Pr[A] \geq \Pr[\overline{A}]$ and hence $\Pr[A] \geq 1/2$.

19.3 BIPARTITE MATCHING

The *matching* problem is one of the canonical optimization problems, arising in all kinds of applications: matching residents with hospitals, kidney donors with patients, flights with crews, and many others. One prototypical variant is *bipartite perfect matching*. In this problem, we are given a bipartite graph $G = (L \cup R, E)$ which has $2n$ vertices partitioned into n-sized sets L and R, where all edges have one endpoint in L and the other in R. The goal is to determine whether there is a *perfect matching*, a subset $M \subseteq E$ of n disjoint edges. That is, M matches every vertex in L to a unique vertex in R.

The bipartite matching problem turns out to have a polynomial-time algorithm, since we can reduce finding a matching in G to find-

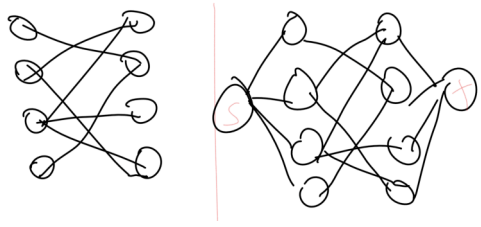


Figure 19.3: The bipartite matching problem in the graph $G = (L \cup R, E)$ can be reduced to the minimum s, t cut problem in the graph G' obtained by adding vertices s, t to G, connecting s with L and connecting t with R.

ing a maximum flow (or equivalently, minimum cut) in a related graph G' (see Fig. 19.3). However, we will see a different probabilistic algorithm to determine whether a graph contains such a matching.

Let us label G's vertices as $L = \{\ell_0, \ldots, \ell_{n-1}\}$ and $R = \{r_0, \ldots, r_{n-1}\}$. A matching M corresponds to a *permutation* $\pi \in S_n$ (i.e., one-to-one and onto function $\pi : [n] \to [n]$) where for every $i \in [n]$, we define $\pi(i)$ to be the unique j such that M contains the edge $\{\ell_i, r_j\}$. Define an $n \times n$ matrix $A = A(G)$ where $A_{i,j} = 1$ if and only if the edge $\{\ell_i, r_j\}$ is present and $A_{i,j} = 0$ otherwise. The correspondence between matchings and permutations implies the following claim:

Lemma 19.6 — Matching polynomial. Define $P = P(G)$ to be the polynomial mapping $\mathbb{R}^{n^2}$ to $\mathbb{R}$ where

$$P(x_{0,0}, \ldots, x_{n-1,n-1}) = \sum_{\pi \in S_n} \left(\prod_{i=0}^{n-1} sign(\pi) A_{i,\pi(i)} \right) \prod_{i=0}^{n-1} x_{i,\pi(i)} \qquad (19.1)$$

Then G has a perfect matching if and only if P is not identically zero. That is, G has a perfect matching if and only if there exists some assignment $x = (x_{i,j})_{i,j \in [n]} \in \mathbb{R}^{n^2}$ such that $P(x) \neq 0$.[4]

Proof. If G has a perfect matching M^*, then let π^* be the permutation corresponding to M and let $x^* \in \mathbb{R}^{n^2}$ defined as follows: $x_{i,j} = 1$ if $j = \pi^*(i)$ and $x^*_{i,j} = 0$ otherwise. (That is, $x^*_{i,j} = 1$ iff $\pi^*(i) = j$.) We claim that $P(x^*) = sign(\pi^*)$ which in particular means that P is not identically zero. To see why this is true, write $P(x^*) = \sum_{\pi \in S_n} sign(\pi) P_\pi(x^*)$ where $P_\pi(x^*) = \prod_{i=0}^{n-1} A_{i,\pi(i)} x^*_{i,\pi(i)}$. But for all $\pi \neq \pi^*$ there will be some i such that $\pi(i) \neq \pi^*(j)$ and so $x^*_{i,\pi(i)} = 0$, which means that $\Pi_\pi(x^*) = 0$. On the other hand, since π^* is a matching in G, $A_{i,\pi^*(i)} = 1$ for all i, and hence $P_{\pi^*}(x^*) = \prod_{i=0}^{n-1} A_{i,\pi^*(i)} x^*_{i,\pi^*(i)} = 1$, and so $P(x^*) = sign(\pi^*)$.

On the other hand, suppose that P is not identically zero. By (19.1), this means there is some $x \in \{0,1\}^{n^2}$ and some permutation π such that $\prod_{i=0}^{n-1} A_{i,\pi(i)} x_{i,\pi(i)} \neq 0$. But for this to happen, it must be that $A_{i,\pi(i)} \neq 0$ for all i, which means that for every i, the edge $(i, \pi(i))$ exists in the graph, and hence π must be a perfect matching in G. ■

As we've seen before, for every $x \in \mathbb{R}^{n^2}$, we can compute $P(x)$ by simply computing the *determinant* of the matrix $A(x)$, which is obtained by replacing $A_{i,j}$ with $A_{i,j} x_{i,j}$. This reduces testing perfect matching to the *zero testing* problem for polynomials: given some polynomial $P(\cdot)$, test whether P is identically zero or not. The intuition behind our randomized algorithm for zero testing is the following:

If a polynomial is not identically zero, then it can't have "too many" roots.

[4] The sign of a permutation $\pi : [n] \to [n]$, denoted by $sign(\pi)$, can be defined in several equivalent ways, one of which is that it equals -1 if the number of pairs $x < y$ s.t. $\pi(x) > \pi(y)$ is odd and equals $+1$ otherwise. The importance of the term $sign(\pi)$ is that it makes P equal to the *determinant* of the matrix $(x_{i,j})$ and hence efficiently computable.

This intuition sort of makes sense. For one variable polynomials, we know that a non-zero linear function has at most one root, a quadratic function (e.g., a parabola) has at most two roots, and generally a degree d equation has at most d roots. While in more than one variable there can be an infinite number of roots (e.g., the polynomial $x_0 + y_0$ vanishes on the line $y = -x$) it is still the case that the set of roots is very "small" compared to the set of all inputs. For example, the root of a bivariate polynomial form a curve, the roots of a three-variable polynomial form a surface, and more generally the roots of an n-variable polynomial are a space of dimension $n - 1$.

This intuition leads to the following simple randomized algorithm:

To decide if P is identically zero, choose a "random" input x and check if $P(x) \neq 0$.

This makes sense: if there are only "few" roots, then we expect that with high probability the random input x is not going to be one of those roots. However, to transform this into an actual algorithm, we need to make both the intuition and the notion of a "random" input precise. Choosing a random real number is quite problematic, especially when you have only a finite number of coins at your disposal, and so we start by reducing the task to a finite setting. We will use the following result:

Theorem 19.7 — Schwartz–Zippel lemma. For every integer q, and polynomial $P : \mathbb{R}^n \to \mathbb{R}$ with integer coefficients. If P has degree at most d and is not identically zero, then it has at most dq^{n-1} roots in the set $[q]^n = \{(x_0, \ldots, x_{n-1}) : x_i \in \{0, \ldots, q-1\}\}$.

We omit the (not too complicated) proof of Theorem 19.7. We remark that it holds not just over the real numbers but over any field as well. Since the matching polynomial P of Lemma 19.6 has degree at most n, Theorem 19.7 leads directly to a simple algorithm for testing if it is non-zero:

Algorithm Perfect-Matching:

Input: Bipartite graph G on $2n$ vertices $\{\ell_0, \ldots, \ell_{n-1}, r_0, \ldots, r_{n-1}\}$.

Operation:

1. For every $i, j \in [n]$, choose $x_{i,j}$ independently at random from $[2n] = \{0, \ldots 2n-1\}$.
2. Compute the determinant of the matrix $A(x)$ whose $(i,j)^{th}$ entry equals $x_{i,j}$ if the edge $\{\ell_i, r_j\}$ is present and 0 otherwise.
3. Output `no perfect matching` if this determinant is zero, and output `perfect matching` otherwise.

This algorithm can be improved further (e.g., see Exercise 19.5). While it is not necessarily faster than the cut-based algorithms for perfect matching, it does have some advantages. In particular, it is more amenable for parallelization. (However, it also has the significant disadvantage that it does not produce a matching but only states that one exists.) The Schwartz–Zippel Lemma, and the associated zero testing algorithm for polynomials, is widely used across computer science, including in several settings where we have no known deterministic algorithm matching their performance.

Chapter Recap

- Using concentration results, we can *amplify* in polynomial time the success probability of a probabilistic algorithm from a mere $1/p(n)$ to $1 - 2^{-q(n)}$ for every polynomials p and q.
- There are several randomized algorithms that are better in various senses (e.g., simpler, faster, or other advantages) than the best known deterministic algorithm for the same problem.

19.4 EXERCISES

Exercise 19.1 — Amplification for max cut. Prove Lemma 19.3 ■

Exercise 19.2 — Deterministic max cut algorithm. [5] ■

[5] TODO: add exercise to give a deterministic max cut algorithm that gives $m/2$ edges. Talk about greedy approach.

Exercise 19.3 — Simulating distributions using coins. Our model for probability involves tossing n coins, but sometimes algorithm require sampling from other distributions, such as selecting a uniform number in $\{0, \ldots, M-1\}$ for some M. Fortunately, we can simulate this with an exponentially small probability of error: prove that for every M, if $n > k\lceil \log M \rceil$, then there is a function $F : \{0,1\}^n \to \{0, \ldots, M-1\} \cup \{\perp\}$ such that **(1)** The probability that $F(x) = \perp$ is at most 2^{-k} and **(2)** the distribution of $F(x)$ conditioned on $F(x) \neq \perp$ is equal to the uniform distribution over $\{0, \ldots, M-1\}$.[6] ■

[6] **Hint:** Think of $x \in \{0,1\}^n$ as choosing k numbers $y_1, \ldots, y_k \in \{0, \ldots, 2^{\lceil \log M \rceil} - 1\}$. Output the first such number that is in $\{0, \ldots, M-1\}$.

Exercise 19.4 — Better walksat analysis.

1. Prove that for every $\epsilon > 0$, if n is large enough then for every $x^* \in \{0,1\}^n$ $\Pr_{x \sim \{0,1\}^n}[\Delta(x, x^*) \leq n/3] \leq 2^{-(1-H(1/3)-\epsilon)n}$ where $H(p) = p\log(1/p) + (1-p)\log(1/(1-p))$ is the same function as in Exercise 18.10.
2. Prove that $2^{1-H(1/4)+(1/4)\log 3} = (3/2)$.
3. Use the above to prove that for every $\delta > 0$ and large enough n, if we set $T = 1000 \cdot (3/2 + \delta)^n$ and $S = n/4$ in the WalkSAT algorithm

then for every satisfiable 3CNF φ, the probability that we output `unsatisfiable` is at most $1/2$.

■

Exercise 19.5 — Faster bipartite matching (challenge). (to be completed: improve the matching algorithm by working modulo a prime)

■

19.5 BIBLIOGRAPHICAL NOTES

The books of Motwani and Raghavan [MR95] and Mitzenmacher and Upfal [MU17] are two excellent resources for randomized algorithms. Some of the history of the discovery of Monte Carlo algorithm is covered here.

19.6 ACKNOWLEDGEMENTS

Learning Objectives:

- **Formal definition of probabilistic polynomial time: the class BPP.**
- **Proof that every function in BPP can be computed by $poly(n)$-sized NAND-CIRC programs/circuits.**
- **Relations between BPP and NP.**
- **Pseudorandom generators**

20
Modeling randomized computation

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin." John von Neumann, 1951.

So far we have described randomized algorithms in an informal way, assuming that an operation such as "pick a string $x \in \{0,1\}^n$" can be done efficiently. We have neglected to address two questions:

1. How do we actually efficiently obtain random strings in the physical world?
2. What is the mathematical model for randomized computations, and is it more powerful than deterministic computation?

The first question is of both practical and theoretical importance, but for now let's just say that there are various physical sources of "random" or "unpredictable" data. A user's mouse movements and typing pattern, (non-solid state) hard drive and network latency, thermal noise, and radioactive decay have all been used as sources for randomness (see discussion in Section 20.8). For example, many Intel chips come with a random number generator built in. One can even build mechanical coin tossing machines (see Fig. 20.1).

This chapter: A non-mathy overview

In this chapter we focus on the second question: formally modeling probabilistic computation and studying its power. We will show that:

1. We can define the class **BPP** that captures all Boolean functions that can be computed in polynomial time by a randomized algorithm. Crucially **BPP** is still very much a *worst case* class of computation: the probability is only over the choice of the random coins of the algorithm, as opposed to the choice of the input.

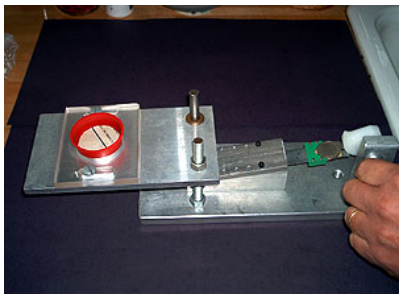


Figure 20.1: A mechanical coin tosser built for Percy Diaconis by Harvard technicians Steve Sansone and Rick Haggerty

2. We can *amplify* the success probability of randomized algorithms, and as a result the definition of the class **BPP** is equivalent whether or not we require $2/3$ success, 0.51 success or every $1 - 2^{-n}$ success.

3. Though, as is the case for **P** and **NP**, there is much we do not know about the class **BPP**, we can establish some relations between **BPP** and the other complexity classes we saw before. In particular we will show that $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{EXP}$ and $\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$.

4. While the relation between **BPP** and **NP** is not known, we can show that if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{BPP} = \mathbf{P}$.

5. We also show that the concept of **NP** completeness applies equally well if we use randomized algorithms as our model of "efficient computation". That is, if a single **NP** complete problem has a randomized polynomial-time algorithm, then all of **NP** can be computed in polynomial-time by randomized algorithms.

6. Finally we will discuss the question of whether $\mathbf{BPP} = \mathbf{P}$ and show some of the intriguing evidence that the answer might actually be *"Yes"* using the concept of *pseudorandom generators*.

20.1 MODELING RANDOMIZED COMPUTATION

Modeling randomized computation is actually quite easy. We can add the following operations to any programming language such as NAND-TM, NAND-RAM, NAND-CIRC etc..:

```
foo = RAND()
```

where `foo` is a variable. The result of applying this operation is that `foo` is assigned a random bit in $\{0,1\}$. (Every time the `RAND` operation is invoked it returns a fresh independent random bit.) We call the programming languages that are augmented with this extra operation RNAND-TM, RNAND-RAM, and RNAND-CIRC respectively.

Similarly, we can easily define randomized Turing machines as Turing machines in which the transition function δ gets as an extra input (in addition to the current state and symbol read from the tape) a bit b that in each step is chosen at random $\in \{0,1\}$. Of course the transition function can ignore this bit (and have the same output regardless of whether $b = 0$ or $b = 1$), and hence randomized Turing machines generalize deterministic Turing machines.

We can use the RAND() operation to define the notion of a function being computed by a randomized $T(n)$ time algorithm for every nice time bound $T : \mathbb{N} \to \mathbb{N}$, as well as the notion of a finite function being computed by a size S randomized NAND-CIRC program (or, equivalently, a randomized circuit with S gates that correspond to either the NAND or coin-tossing operations). However, for simplicity we will not define randomized computation in full generality, but simply focus on the class of functions that are computable by randomized algorithms *running in polynomial time*, which by historical convention is known as **BPP**:

Definition 20.1 — The class BPP. Let $F : \{0,1\}^* \to \{0,1\}$. We say that $F \in$ **BPP** if there exist constants $a, b \in \mathbb{N}$ and an RNAND-TM program P such that for every $x \in \{0,1\}^*$, on input x, the program P halts within at most $a|x|^b$ steps and

$$\Pr[P(x) = F(x)] \geq \tfrac{2}{3} \tag{20.1}$$

where this probability is taken over the result of the RAND operations of P.

Note that the probability in (20.1) is taken only over the random choices in the execution of P and *not* over the choice of the input x. In particular, as discussed in Big Idea 24, **BPP** is still a *worst case* complexity class, in the sense that if F is in **BPP** then there is a polynomial-time randomized algorithm that computes F with probability at least $2/3$ *on every possible* (and not just random) input.

The same polynomial-overhead simulation of NAND-RAM programs by NAND-TM programs we saw in Theorem 13.5 extends to *randomized* programs as well. Hence the class **BPP** is the same regardless of whether it is defined via RNAND-TM or RNAND-RAM programs. Similarly, we could have just as well defined **BPP** using randomized Turing machines.

Because of these equivalences, below we will use the name *"polynomial time randomized algorithm"* to denote a computation that can be modeled by a polynomial-time RNAND-TM program, RNAND-RAM program, or a randomized Turing machine (or any programming language that includes a coin tossing operation). Since all these models are equivalent up to polynomial factors, you can use your favorite model to capture polynomial-time randomized algorithms without any loss in generality.

Solved Exercise 20.1 — Choosing from a set. Modern programming languages often involve not just the ability to toss a random coin in $\{0,1\}$ but also to choose an element at random from a set S. Show that you can emulate this primitive using coin tossing. Specifically, show that

there is a randomized algorithm A that, on input a set S of m strings of length n, runs in time $poly(n, m)$ and outputs either an element $x \in S$ or "fail" such that

1. Let p be the probability that A outputs "fail", then $p < 2^{-n}$ (a number small enough that it can be ignored).
2. For every $x \in S$, the probability that A outputs x is exactly $\frac{1-p}{m}$ (and so the output is uniform over S if we ignore the tiny probability of failure)

■

Solution:

If the size of S is a power of two, that is $m = 2^\ell$ for some $\ell \in N$, then we can choose a random element in S by tossing ℓ coins to obtain a string $w \in \{0,1\}^\ell$ and then output the i-th element of S where i is the number whose binary representation is w.

If S is not a power of two, then our first attempt will be to let $\ell = \lceil \log m \rceil$ and do the same, but then output the i-th element of S if $i \in [m]$ and output "fail" otherwise. Conditioned on not outputting "fail", this element is distributed uniformly in S. However, in the worst case, 2^ℓ can be almost $2m$ and so the probability of fail might be close to half. To reduce the failure probability, we can repeat the experiment above n times. Specifically, we will use the following algorithm

Algorithm 20.2 — Sample from set.

Input: Set $S = \{x_0, \ldots, x_{m-1}\}$ with $x_i \in \{0,1\}^n$ for all $i \in [m]$.
Output: Either $x \in S$ or "fail"

```
Let ℓ ← ⌈log m⌉
for j = 0, 1, ..., n − 1 do
    Pick w ∼ {0,1}^ℓ
    Let i ∈ [2^ℓ] be number whose binary representation is w.
    if i < m then
        return x_i
    end if
end for
return "fail"
```

Conditioned on not failing, the output of Algorithm 20.2 is uniformly distributed in S. However, since $2^\ell < 2m$, the probability

of failure in each iteration is less than $1/2$ and so the probability of failure in all of them is at most $(1/2)^n = 2^{-n}$.

■

20.1.1 An alternative view: random coins as an "extra input"

While we presented randomized computation as adding an extra "coin tossing" operation to our programs, we can also model this as being given an additional extra input. That is, we can think of a randomized algorithm A as a *deterministic* algorithm A' that takes *two inputs* x and r where the second input r is chosen at random from $\{0,1\}^m$ for some $m \in \mathbb{N}$ (see Fig. 20.2). The equivalence to the Definition 20.1 is shown in the following theorem:

Theorem 20.3 — Alternative characterization of BPP. Let $F : \{0,1\}^* \to \{0,1\}$. Then $F \in$ **BPP** if and only if there exists $a, b \in \mathbb{N}$ and $G : \{0,1\}^* \to \{0,1\}$ such that G is in **P** and for every $x \in \{0,1\}^*$,

$$\Pr_{r \sim \{0,1\}^{a|x|^b}}[G(xr) = F(x)] \geq \tfrac{2}{3} \,. \tag{20.2}$$

Proof Idea:

The idea behind the proof is that, as illustrated in Fig. 20.2, we can simply replace sampling a random coin with reading a bit from the extra "random input" r and vice versa. To prove this rigorously we need to work through some slightly cumbersome formal notation. This might be one of those proofs that is easier to work out on your own than to read.

⋆

Proof of Theorem 20.3. We start by showing the "only if" direction. Let $F \in$ **BPP** and let P be an RNAND-TM program that computes F as per Definition 20.1, and let $a, b \in \mathbb{N}$ be such that on every input of length n, the program P halts within at most an^b steps. We will construct a polynomial-time algorithm P' such that for every $x \in \{0,1\}^n$, if we set $m = an^b$, then

$$\Pr_{r \sim \{0,1\}^m}[P'(xr) = 1] = \Pr[P(x) = 1] \,,$$

where the probability in the right-hand side is taken over the `RAND()` operations in P. In particular this means that if we define $G(xr) = P'(xr)$ then the function G satisfies the conditions of (20.2).

The algorithm P' will be very simple: it simulates the program P, maintaining a counter i initialized to 0. Every time that P makes a `RAND()` operation, the program P' will supply the result from r_i and increment i by one. We will never "run out" of bits, since the running time of P is at most an^b and hence it can make at most this number of

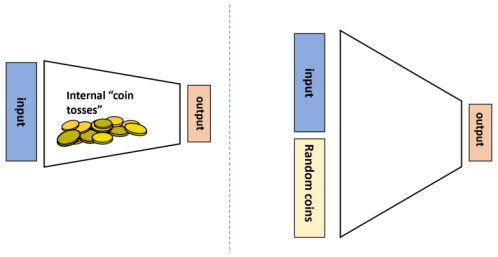


Figure 20.2: The two equivalent views of randomized algorithms. We can think of such an algorithm as having access to an internal `RAND()` operation that outputs a random independent value in $\{0,1\}$ whenever it is invoked, or we can think of it as a deterministic algorithm that in addition to the standard input $x \in \{0,1\}^n$ obtains an additional auxiliary input $r \in \{0,1\}^m$ that is chosen uniformly at random.

`RAND()` calls. The output of $P'(xr)$ for a random $r \sim \{0,1\}^m$ will be distributed identically to the output of $P(x)$.

For the other direction, given a function $G \in$ **P** satisfying the condition (20.2) and a NAND-TM P' that computes G in polynomial time, we can construct an RNAND-TM program P that computes F in polynomial time. On input $x \in \{0,1\}^n$, the program P will simply use the `RAND()` instruction an^b times to fill an array `R[0]` , ..., `R[`$an^b - 1$`]` and then execute the original program P' on input xr where r_i is the i-th element of the array `R`. Once again, it is clear that if P' runs in polynomial time then so will P, and for every input x and $r \in \{0,1\}^{an^b}$, the output of P on input x and where the coin tosses outcome is r is equal to $P'(xr)$.

■

R

Remark 20.4 — Definitions of BPP and NP. The characterization of **BPP** in Theorem 20.3 is reminiscent of the characterization of **NP** in Definition 15.1, with the randomness in the case of **BPP** playing the role of the solution in the case of **NP**. However, there are important differences between the two:

- The definition of **NP** is "one sided": $F(x) = 1$ if *there exists* a solution w such that $G(xw) = 1$ and $F(x) = 0$ if *for every* string w of the appropriate length, $G(xw) = 0$. In contrast, the characterization of **BPP** is symmetric with respect to the cases $F(x) = 0$ and $F(x) = 1$.
- The relation between **NP** and **BPP** is not immediately clear. It is not known whether **BPP** $\subseteq$ **NP**, **NP** $\subseteq$ **BPP**, or these two classes are incomparable. It is however known (with a non-trivial proof) that if **P** $=$ **NP** then **BPP** $=$ **P** (see Theorem 20.11).
- Most importantly, the definition of **NP** is "ineffective," since it does not yield a way of actually finding whether there exists a solution among the exponentially many possibilities. By contrast, the definition of **BPP** gives us a way to compute the function in practice by simply choosing the second input at random.

"Random tapes". Theorem 20.3 motivates sometimes considering the randomness of an RNAND-TM (or RNAND-RAM) program as an extra input. As such, if A is a randomized algorithm that on inputs of length n makes at most m coin tosses, we will often use the notation $A(x;r)$ (where $x \in \{0,1\}^n$ and $r \in \{0,1\}^m$) to refer to the result of executing x when the coin tosses of A correspond to the coordinates of r. This second, or "auxiliary," input is sometimes referred to as

a "random tape." This terminology originates from the model of randomized Turing machines.

20.1.2 Success amplification of two-sided error algorithms

The number $2/3$ might seem arbitrary, but as we've seen in Chapter 19 it can be amplified to our liking:

Theorem 20.5 — Amplification. Let $F : \{0,1\}^* \to \{0,1\}$ be a Boolean function such that there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time randomized algorithm A satisfying that for every $x \in \{0,1\}^n$,

$$\Pr[A(x) = F(x)] \geq \frac{1}{2} + \frac{1}{p(n)} . \tag{20.3}$$

Then for every polynomial $q : \mathbb{N} \to \mathbb{N}$ there is a polynomial-time randomized algorithm B satisfying for every $x \in \{0,1\}^n$,

$$\Pr[B(x) = F(x)] \geq 1 - 2^{-q(n)} .$$

Big Idea 25 We can *amplify* the success of randomized algorithms to a value that is arbitrarily close to 1.

Proof Idea:

The proof is the same as we've seen before in the case of maximum cut and other examples. We use the Chernoff bound to argue that if A computes F with probability at least $\frac{1}{2} + \epsilon$ and we run it $O(k/\epsilon^2)$ times, each time using fresh and independent random coins, then the probability that the majority of the answers will not be correct will be less than 2^{-k}. Amplification can be thought of as a "polling" of the choices for randomness for the algorithm (see Fig. 20.3).

⋆

Proof of Theorem 20.5. Let A be an algorithm satisfying (20.3). Set $\epsilon = \frac{1}{p(n)}$ and $k = q(n)$ where p, q are the polynomials in the theorem statement. We can run P on input x for $t = 10k/\epsilon^2$ times, using fresh randomness in each execution, and compute the outputs $y_0, \ldots, y_{t-1}$. We output the value y that appeared the largest number of times. Let X_i be the random variable that is equal to 1 if $y_i = F(x)$ and equal to 0 otherwise. The random variables $X_0, \ldots, X_{t-1}$ are i.i.d. and satisfy $\mathbb{E}[X_i] = \Pr[X_i = 1] \geq 1/2 + \epsilon$, and hence by linearity of expectation $\mathbb{E}[\sum_{i=0}^{t-1} X_i] \geq t(1/2 + \epsilon)$. For the plurality value to be *incorrect*, it must hold that $\sum_{i=0}^{t-1} X_i \leq t/2$, which means that $\sum_{i=0}^{t-1} X_i$ is at least ϵt far from its expectation. Hence by the Chernoff bound (Theorem 18.12),

the probability that the plurality value is not correct is at most $2e^{-\epsilon^2 t}$, which is smaller than 2^{-k} for our choice of t.

■

20.2 BPP AND NP COMPLETENESS

Since "noisy processes" abound in nature, randomized algorithms can be realized physically, and so it is reasonable to propose **BPP** rather than **P** as our mathematical model for "feasible" or "tractable" computation. One might wonder if this makes all the previous chapters irrelevant, and in particular if the theory of **NP** completeness still applies to probabilistic algorithms. Fortunately, the answer is *Yes*:

Theorem 20.6 — NP hardness and BPP. Suppose that F is **NP**-hard and $F \in$ **BPP**. Then **NP** $\subseteq$ **BPP**.

Before seeing the proof, note that Theorem 20.6 implies that if there was a randomized polynomial time algorithm for any **NP**-complete problem such as $3SAT$, *ISET* etc., then there would be such an algorithm for *every* problem in **NP**. Thus, regardless of whether our model of computation is deterministic or randomized algorithms, **NP** complete problems retain their status as the "hardest problems in **NP**."

Proof Idea:

The idea is to simply run the reduction as usual, and plug it into the randomized algorithm instead of a deterministic one. It would be an excellent exercise, and a way to reinforce the definitions of **NP**-hardness and randomized algorithms, for you to work out the proof for yourself. However for the sake of completeness, we include this proof below.

⋆

Proof of Theorem 20.6. Suppose that F is **NP**-hard and $F \in$ **BPP**. We will now show that this implies that **NP** $\subseteq$ **BPP**. Let $G \in$ **NP**. By the definition of **NP**-hardness, it follows that $G \leq_p F$, or that in other words there exists a polynomial-time computable function $R : \{0,1\}^* \to \{0,1\}^*$ such that $G(x) = F(R(x))$ for every $x \in \{0,1\}^*$. Now if F is in **BPP** then there is a polynomial-time RNAND-TM program P such that

$$\Pr[P(y) = F(y)] \geq 2/3 \tag{20.4}$$

for *every* $y \in \{0,1\}^*$ (where the probability is taken over the random coin tosses of P). Hence we can get a polynomial-time RNAND-TM program P' to compute G by setting $P'(x) = P(R(x))$. By (20.4) $\Pr[P'(x) = F(R(x))] \geq 2/3$ and since $F(R(x)) = G(x)$ this implies that $\Pr[P'(x) = G(x)] \geq 2/3$, which proves that $G \in$ **BPP**.

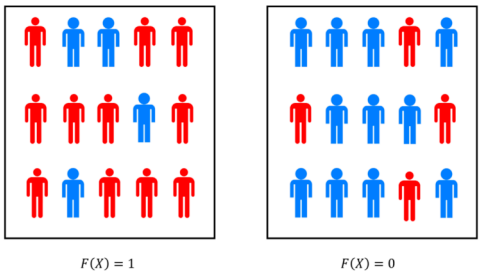


Figure 20.3: If $F \in$ **BPP** then there is a randomized polynomial-time algorithm P with the following property: In the case $F(x) = 0$ two thirds of the "population" of random choices satisfy $P(x;r) = 0$ and in the case $F(x) = 1$ two thirds of the population satisfy $P(x;r) = 1$. We can think of amplification as a form of "polling" of the choices of randomness. By the Chernoff bound, if we poll a sample of $O(\frac{\log(1/\delta)}{\epsilon^2})$ random choices r, then with probability at least $1-\delta$, the fraction of r's in the sample satisfying $P(x;r) = 1$ will give us an estimate of the fraction of the population within an ϵ margin of error. This is the same calculation used by pollsters to determine the needed sample size in their polls.

■

Most of the results we've seen about **NP** hardness, including the search to decision reduction of Theorem 16.1, the decision to optimization reduction of Theorem 16.3, and the quantifier elimination result of Theorem 16.6, all carry over in the same way if we replace **P** with **BPP** as our model of efficient computation. Thus if **NP** $\subseteq$ **BPP** then we get essentially all of the strange and wonderful consequences of **P** $=$ **NP**. Unsurprisingly, we cannot rule out this possibility. In fact, unlike **P** $=$ **EXP**, which is ruled out by the time hierarchy theorem, we don't even know how to rule out the possibility that **BPP** $=$ **EXP**! Thus a priori it's possible (though seems highly unlikely) that randomness is a magical tool that allows us to speed up arbitrary exponential time computation.[1] Nevertheless, as we discuss below, it is believed that randomization's power is much weaker and **BPP** lies in much more "pedestrian" territory.

[1] At the time of this writing, the largest "natural" complexity class which we can't rule out being contained in **BPP** is the class **NEXP**, which we did not define in this course, but corresponds to non-deterministic exponential time. See this paper for a discussion of this question.

20.3 THE POWER OF RANDOMIZATION

A major question is whether randomization can add power to computation. Mathematically, we can phrase this as the following question: does **BPP** $=$ **P**? Given what we've seen so far about the relations of other complexity classes such as **P** and **NP**, or **NP** and **EXP**, one might guess that:

1. We do not know the answer to this question.

2. But we suspect that **BPP** is different than **P**.

One would be correct about the former, but wrong about the latter. As we will see, we do in fact have reasons to believe that **BPP** $=$ **P**. This can be thought of as supporting the *extended Church Turing hypothesis* that deterministic polynomial-time Turing machines capture what can be feasibly computed in the physical world.

We now survey some of the relations that are known between **BPP** and other complexity classes we have encountered. (See also Fig. 20.4.)

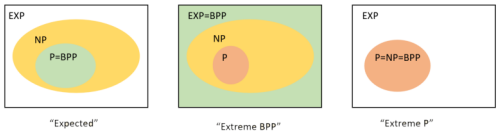


Figure 20.4: Some possibilities for the relations between **BPP** and other complexity classes. Most researchers believe that **BPP** $=$ **P** and that these classes are *not* powerful enough to solve **NP**-complete problems, let alone all problems in **EXP**. However, we have not even been able yet to rule out the possibility that randomness is a "silver bullet" that allows exponential speedup on all problems, and hence **BPP** $=$ **EXP**. As we've already seen, we also can't rule out that **P** $=$ **NP**. Interestingly, in the latter case, **P** $=$ **BPP**.

20.3.1 Solving BPP in exponential time

It is not hard to see that if F is in **BPP** then it can be computed in *exponential* time.

Theorem 20.7 — Simulating randomized algorithms in exponential time.
BPP $\subseteq$ **EXP**

P

The proof of Theorem 20.7 readily follows by enumerating over all the (exponentially many) choices for the random coins. We omit the formal proof, as doing it by yourself is an excellent way to get comfortable with Definition 20.1.

20.3.2 Simulating randomized algorithms by circuits

We have seen in Theorem 13.12 that if F is in **P**, then there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every n, the restriction $F_{\restriction n}$ of F to inputs $\{0,1\}^n$ is in $SIZE(p(n))$. (In other words, that $\mathbf{P} \subseteq \mathbf{P_{/poly}}$.) A priori it is not at all clear that the same holds for a function in **BPP**, but this does turn out to be the case.

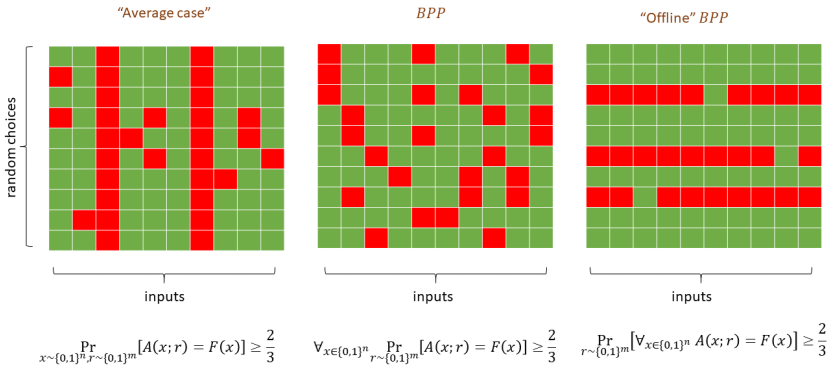


Figure 20.5: The possible guarantees for a randomized algorithm A computing some function F. In the tables above, the columns correspond to different inputs and the rows to different choices of the random tape. A cell at position r, x is colored green if $A(x;r) = F(x)$ (i.e., the algorithm outputs the correct answer) and red otherwise. The standard **BPP** guarantee corresponds to the middle figure, where for every input x, at least two thirds of the choices r for a random tape will result in A computing the correct value. That is, every column is colored green in at least two thirds of its coordinates. In the left figure we have an "average case" guarantee where the algorithm is only guaranteed to output the correct answer with probability two thirds over a *random* input (i.e., at most one third of the total entries of the table are colored red, but there could be an all red column). The right figure corresponds to the "offline **BPP**" case, with probability at least two thirds over the random choice r, r will be good for *every* input. That is, at least two thirds of the rows are all green. Theorem 20.8 ($\mathbf{BPP} \subseteq \mathbf{P_{/poly}}$) is proven by amplifying the success of a **BPP** algorithm until we have the "offline **BPP**" guarantee, and then hardwiring the choice of the randomness r to obtain a non-uniform deterministic algorithm.

Theorem 20.8 — Randomness does not help for non-uniform computation.
$\mathbf{BPP} \subseteq \mathbf{P_{/poly}}$.

That is, for every $F \in \mathbf{BPP}$, there exist some $a, b \in \mathbb{N}$ such that for every $n > 0$, $F_{\restriction n} \in SIZE(an^b)$ where $F_{\restriction n}$ is the restriction of F to inputs in $\{0,1\}^n$.

Proof Idea:

The idea behind the proof is that we can first amplify by repetition the probability of success from $2/3$ to $1 - 0.1 \cdot 2^{-n}$. This will allow us to show that for every $n \in \mathbb{N}$ there exists a *single fixed choice* of "favorable coins" which is a string r of length polynomial in n such that if r is used for the randomness then we output the right answer on *all* of the possible 2^n inputs. We can then use the standard "unravelling the loop" technique to transform an RNAND-TM program to an RNAND-CIRC program, and "hardwire" the favorable choice of random coins

to transform the RNAND-CIRC program into a plain old deterministic NAND-CIRC program.

⋆

Proof of Theorem 20.8. Suppose that $F \in \mathbf{BPP}$. Let P be a polynomial-time RNAND-TM program that computes F as per Definition 20.1. Using Theorem 20.5, we can *amplify* the success probability of P to obtain an RNAND-TM program P' that is at most a factor of $O(n)$ slower (and hence still polynomial time) such that for every $x \in \{0,1\}^n$

$$\Pr_{r\sim\{0,1\}^m}[P'(x;r) = F(x)] \geq 1 - 0.1 \cdot 2^{-n} \,, \tag{20.5}$$

where m is the number of coin tosses that P' uses on inputs of length n. We use the notation $P'(x;r)$ to denote the execution of P' on input x and when the result of the coin tosses corresponds to the string r.

For every $x \in \{0,1\}^n$, define the "bad" event B_x to hold if $P'(x) \neq F(x)$, where the sample space for this event consists of the coins of P'. Then by (20.5), $\Pr[B_x] \leq 0.1 \cdot 2^{-n}$ for every $x \in \{0,1\}^n$. Since there are 2^n many such x's, by the union bound we see that the probability that the *union* of the events $\{B_x\}_{x\in\{0,1\}^n}$ is at most 0.1. This means that if we choose $r \sim \{0,1\}^m$, then with probability at least 0.9 it will be the case that for *every* $x \in \{0,1\}^n$, $F(x) = P'(x;r)$. (Indeed, otherwise the event B_x would hold for some x.) In particular, because of the mere fact that the probability of $\cup_{x\in\{0,1\}^n} B_x$ is smaller than 1, this means that *there exists* a particular $r^* \in \{0,1\}^m$ such that

$$P'(x;r^*) = F(x) \tag{20.6}$$

for every $x \in \{0,1\}^n$.

Now let us use the standard "unravelling the loop" technique and transform P' into a NAND-CIRC program Q of polynomial in n size, such that $Q(xr) = P'(x;r)$ for every $x \in \{0,1\}^n$ and $r \in \{0,1\}^m$. Then by "hardwiring" the values $r^*_0, \ldots, r^*_{m-1}$ in place of the last m inputs of Q, we obtain a new NAND-CIRC program Q_{r^*} that satisfies by (20.6) that $Q_{r^*}(x) = F(x)$ for every $x \in \{0,1\}^n$. This demonstrates that $F_{\restriction n}$ has a polynomial-sized NAND-CIRC program, hence completing the proof of Theorem 20.8.

■

20.4 DERANDOMIZATION

The proof of Theorem 20.8 can be summarized as follows: we can replace a $poly(n)$-time algorithm that tosses coins as it runs with an algorithm that uses a single set of coin tosses $r^* \in \{0,1\}^{poly(n)}$ which

will be good enough for all inputs of size n. Another way to say it is that for the purposes of computing functions, we do not need "online" access to random coins and can generate a set of coins "offline" ahead of time, before we see the actual input.

But this does not really help us with answering the question of whether **BPP** equals **P**, since we still need to find a way to generate these "offline" coins in the first place. To derandomize an RNAND-TM program we will need to come up with a *single* deterministic algorithm that will work for *all input lengths*. That is, unlike in the case of RNAND-CIRC programs, we cannot choose for every input length n some string $r^* \in \{0,1\}^{poly(n)}$ to use as our random coins.

Can we derandomize randomized algorithms, or does randomness add an inherent extra power for computation? This is a fundamentally interesting question but is also of practical significance. Ever since people started to use randomized algorithms during the Manhattan project, they have been trying to remove the need for randomness and replace it with numbers that are selected through some deterministic process. Throughout the years this approach has often been used successfully, though there have been a number of failures as well.[2]

A common approach people used over the years was to replace the random coins of the algorithm by a "randomish looking" string that they generated through some arithmetic progress. For example, one can use the digits of π for the random tape. Using these type of methods corresponds to what von Neumann referred to as a "state of sin". (Though this is a sin that he himself frequently committed, as generating true randomness in sufficient quantity was and still is often too expensive.) The reason that this is considered a "sin" is that such a procedure will not work in general. For example, it is easy to modify any probabilistic algorithm A such as the ones we have seen in Chapter 19, to an algorithm A' that is *guaranteed to fail* if the random tape happens to equal the digits of π. This means that the procedure "replace the random tape by the digits of π" does not yield a *general* way to transform a probabilistic algorithm to a deterministic one that will solve the same problem. Of course, this procedure does not *always* fail, but we have no good way to determine when it fails and when it succeeds. This reasoning is not specific to π and holds for every deterministically produced string, whether it obtained by π, e, the Fibonacci series, or anything else.

An algorithm that checks if its random tape is equal to π and then fails seems to be quite silly, but this is but the "tip of the iceberg" for a very serious issue. Time and again people have learned the hard way that one needs to be very careful about producing random bits using deterministic means. As we will see when we discuss cryptography,

[2] One amusing anecdote is a recent case where scammers managed to predict the imperfect "pseudo-random generator" used by slot machines to cheat casinos. Unfortunately we don't know the details of how they did it, since the case was sealed.

many spectacular security failures and break-ins were the result of using "insufficiently random" coins.

20.4.1 Pseudorandom generators

So, we can't use any *single* string to "derandomize" a probabilistic algorithm. It turns out however, that we can use a *collection* of strings to do so. Another way to think about it is that rather than trying to *eliminate* the need for randomness, we start by focusing on *reducing* the amount of randomness needed. (Though we will see that if we reduce the randomness sufficiently, we can eventually get rid of it altogether.)

We make the following definition:

Definition 20.9 — Pseudorandom generator. A function $G : \{0,1\}^\ell \to \{0,1\}^m$ is a (T,ϵ)-*pseudorandom generator* if for every circuit C with m inputs, one output, and at most T gates,

$$\left| \Pr_{s\sim\{0,1\}^\ell}[C(G(s)) = 1] - \Pr_{r\sim\{0,1\}^m}[C(r) = 1] \right| < \epsilon \tag{20.7}$$

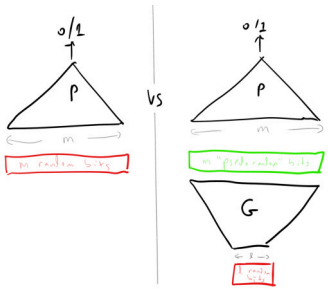


Figure 20.6: A pseudorandom generator G maps a short string $s \in \{0,1\}^\ell$ into a long string $r \in \{0,1\}^m$ such that a small program/circuit P cannot distinguish between the case that it is provided a random input $r \sim \{0,1\}^m$ and the case that it is provided a "pseudorandom" input of the form $r = G(s)$ where $s \sim \{0,1\}^\ell$. The short string s is sometimes called the *seed* of the pseudorandom generator, as it is a small object that can be thought as yielding a large "tree of randomness".

This is a definition that's worth reading more than once, and spending some time to digest it. Note that it takes several parameters:

- T is the limit on the number of gates of the circuit C that the generator needs to "fool". The larger T is, the stronger the generator.
- ϵ is how close the output of the pseudorandom generator is to the true uniform distribution over $\{0,1\}^m$. The smaller ϵ is, the stronger the generator.
- ℓ is the input length and m is the output length. If $\ell \geq m$ then it is trivial to come up with such a generator: on input $s \in \{0,1\}^\ell$, we can output $s_0, \ldots, s_{m-1}$. In this case $\Pr_{s\sim\{0,1\}^\ell}[P(G(s)) = 1]$ will simply equal $\Pr_{r\sim\{0,1\}^m}[P(r) = 1]$, no matter how many lines P has. So, the smaller ℓ is and the larger m is, the stronger the generator, and to get anything non-trivial, we need $m > \ell$.

Furthermore note that although our eventual goal is to fool probabilistic randomized algorithms that take an unbounded number of inputs, Definition 20.9 refers to *finite* and *deterministic* NAND-CIRC programs.

We can think of a pseudorandom generator as a "randomness amplifier." It takes an input s of ℓ bits chosen at random and expands these ℓ bits into an output r of $m > \ell$ *pseudorandom* bits. If

ϵ is small enough then the pseudorandom bits will "look random" to any NAND-CIRC program that is not too big. Still, there are two questions we haven't answered:

- *What reason do we have to believe that pseudorandom generators with non-trivial parameters exist?*
- *Even if they do exist, why would such generators be useful to derandomize randomized algorithms?* After all, Definition 20.9 does not involve RNAND-TM or RNAND-RAM programs, but rather deterministic NAND-CIRC programs with no randomness and no loops.

We will now (partially) answer both questions. For the first question, let us come clean and confess we do not know how to *prove* that interesting pseudorandom generators exist. By *interesting* we mean pseudorandom generators that satisfy that ϵ is some small constant (say $\epsilon < 1/3$), $m > \ell$, and the function G itself can be computed in $poly(m)$ time. Nevertheless, Lemma 20.12 (whose statement and proof is deferred to the end of this chapter) shows that if we only drop the last condition (polynomial-time computability), then there do in fact exist pseudorandom generators where m is *exponentially larger* than ℓ.

At this point you might want to skip ahead and look at the *statement* of Lemma 20.12. However, since its *proof* is somewhat subtle, I recommend you defer reading it until you've finished reading the rest of this chapter.

20.4.2 From existence to constructivity

The fact that there *exists* a pseudorandom generator does not mean that there is one that can be efficiently computed. However, it turns out that we can turn complexity "on its head" and use the assumed *non-existence* of fast algorithms for problems such as 3SAT to obtain pseudorandom generators that can then be used to transform randomized algorithms into deterministic ones. This is known as the *Hardness vs Randomness* paradigm. A number of results along those lines, most of which are outside the scope of this course, have led researchers to believe the following conjecture:

Optimal PRG conjecture: There is a polynomial-time computable function $PRG : \{0,1\}^* \rightarrow \{0,1\}$ that yields an *exponentially secure pseudorandom generator*.

Specifically, there exists a constant $\delta > 0$ such that for every ℓ and $m < 2^{\delta \ell}$, if we define $G : \{0,1\}^\ell \rightarrow \{0,1\}^m$ as $G(s)_i = PRG(s,i)$ for every $s \in \{0,1\}^\ell$ and $i \in [m]$, then G is a $(2^{\delta\ell}, 2^{-\delta\ell})$ pseudorandom generator.

P

The "optimal PRG conjecture" is worth while reading more than once. What it posits is that we can obtain a (T, ϵ) pseudorandom generator G such that every output bit of G can be computed in time polynomial in the length ℓ of the input, where T is exponentially large in ℓ and ϵ is exponentially small in ℓ. (Note that we could not hope for the entire output to be computable in ℓ, as just writing the output down will take too long.)

To understand why we call such a pseudorandom generator "optimal," it is a great exercise to convince yourself that, for example, there does not exist a $(2^{1.1\ell}, 2^{-1.1\ell})$ pseudorandom generator (in fact, the number δ in the conjecture must be smaller than 1). To see that we can't have $T \gg 2^\ell$, note that if we allow a NAND-CIRC program with much more than 2^ℓ lines then this NAND-CIRC program could "hardwire" inside it all the outputs of G on all its 2^ℓ inputs, and use that to distinguish between a string of the form $G(s)$ and a uniformly chosen string in $\{0,1\}^m$. To see that we can't have $\epsilon \ll 2^{-\ell}$, note that by guessing the input s (which will be successful with probability $2^{-2\ell}$), we can obtain a small (i.e., $O(\ell)$ line) NAND-CIRC program that achieves a $2^{-\ell}$ advantage in distinguishing a pseudorandom and uniform input. Working out these details is a highly recommended exercise.

We emphasize again that the optimal PRG conjecture is, as its name implies, a *conjecture*, and we still do not know how to *prove* it. In particular, it is stronger than the conjecture that $\mathbf{P} \neq \mathbf{NP}$. But we do have some evidence for its truth. There is a spectrum of different types of pseudorandom generators, and there are weaker assumptions than the optimal PRG conjecture that suffice to prove that $\mathbf{BPP} = \mathbf{P}$. In particular this is known to hold under the assumption that there exists a function $F \in \mathbf{TIME}(2^{O(n)})$ and $\epsilon > 0$ such that for every sufficiently large n, $F_{\restriction n}$ is not in $SIZE(2^{\epsilon n})$. The name "Optimal PRG conjecture" is non-standard. This conjecture is sometimes known in the literature as the existence of *exponentially strong pseudorandom functions*.[3]

[3] A pseudorandom generator of the form we posit, where each output bit can be computed individually in time polynomial in the seed length, is commonly known as a *pseudorandom function generator*. For more on the many interesting results and connections in the study of *pseudorandomness*, see this monograph of Salil Vadhan.

20.4.3 Usefulness of pseudorandom generators

We now show that optimal pseudorandom generators are indeed very useful, by proving the following theorem:

Theorem 20.10 — Derandomization of BPP. Suppose that the optimal PRG conjecture is true. Then $\mathbf{BPP} = \mathbf{P}$.

Proof Idea:

The optimal PRG conjecture tells us that we can achieve *exponential expansion* of ℓ truly random coins into as many as $2^{\delta\ell}$ "pseudorandom coins." Looked at from the other direction, it allows us to reduce the need for randomness by taking an algorithm that uses m coins and converting it into an algorithm that only uses $O(\log m)$ coins. Now an algorithm of the latter type by can be made fully deterministic by enumerating over all the $2^{O(\log m)}$ (which is polynomial in m) possibilities for its random choices.

⋆

We now proceed with the proof details.

Proof of Theorem 20.10. Let $F \in$ **BPP** and let P be a NAND-TM program and a, b, c, d constants such that for every $x \in \{0,1\}^n$, $P(x)$ runs in at most $c \cdot n^d$ steps and $\Pr_{r\sim\{0,1\}^m}[P(x;r) = F(x)] \geq 2/3$. By "unrolling the loop" and hardwiring the input x, we can obtain for every input $x \in \{0,1\}^n$ a NAND-CIRC program Q_x of at most, say, $T = 10c \cdot n^d$ lines, that takes m bits of input and such that $Q(r) = P(x;r)$.

Now suppose that $G : \{0,1\}^\ell \to \{0,1\}$ is a $(T, 0.1)$ pseudorandom generator. Then we could deterministically estimate the probability $p(x) = \Pr_{r\sim\{0,1\}^m}[Q_x(r) = 1]$ up to 0.1 accuracy in time $O(T \cdot 2^\ell \cdot m \cdot cost(G))$ where $cost(G)$ is the time that it takes to compute a single output bit of G.

The reason is that we know that $\tilde{p}(x) = \Pr_{s\sim\{0,1\}^\ell}[Q_x(G(s)) = 1]$ will give us such an estimate for $p(x)$, and we can compute the probability $\tilde{p}(x)$ by simply trying all 2^ℓ possibillites for s. Now, under the optimal PRG conjecture we can set $T = 2^{\delta\ell}$ or equivalently $\ell = \frac{1}{\delta}\log T$, and our total computation time is polynomial in $2^\ell = T^{1/\delta}$. Since $T \leq 10c \cdot n^d$, this running time will be polynomial in n.

This completes the proof, since we are guaranteed that $\Pr_{r\sim\{0,1\}^m}[Q_x(r) = F(x)] \geq 2/3$, and hence estimating the probability $p(x)$ to within 0.1 accuracy is sufficient to compute $F(x)$. ■

20.5 P = NP AND BPP VS P

Two computational complexity questions that we cannot settle are:

- Is **P** = **NP**? Where we believe the answer is *negative*.
- Is **BPP** = **P**? Where we believe the answer is *positive*.

However we can say that the "conventional wisdom" is correct on at least one of these questions. Namely, if we're wrong on the first count, then we'll be right on the second one:

Theorem 20.11 — Sipser–Gács Theorem. If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{BPP} = \mathbf{P}$.

Before reading the proof, it is instructive to think why this result is not "obvious." If $\mathbf{P} = \mathbf{NP}$ then given any randomized algorithm A and input x, we will be able to figure out in polynomial time if there is a string $r \in \{0,1\}^m$ of random coins for A such that $A(xr) = 1$. The problem is that even if $\Pr_{r\sim\{0,1\}^m}[A(xr) = F(x)] \geq 0.9999$, it can still be the case that even when $F(x) = 0$ there exists a string r such that $A(xr) = 1$.

The proof is rather subtle. It is much more important that you understand the *statement* of the theorem than that you follow all the details of the proof.

Proof Idea:

The construction follows the "quantifier elimination" idea which we have seen in Theorem 16.6. We will show that for every $F \in \mathbf{BPP}$, we can reduce the question of some input x satisfies $F(x) = 1$ to the question of whether a formula of the form $\exists_{u\in\{0,1\}^m} \forall_{v\in\{0,1\}^k} P(u,v)$ is true, where m, k are polynomial in the length of x and P is polynomial-time computable. By Theorem 16.6, if $\mathbf{P} = \mathbf{NP}$ then we can decide in polynomial time whether such a formula is true or false.

The idea behind this construction is that using amplification we can obtain a randomized algorithm A for computing F using m coins such that for every $x \in \{0,1\}^n$, if $F(x) = 0$ then the set $S \subseteq \{0,1\}^m$ of coins that make A output 1 is extremely tiny (i.e., exponentially small relative to 2^m), and if $F(x) = 1$ then S is very large (of size close to 2^m). We then consider "shifts" of the set S: sets of the form $S \oplus s$ where $s \in \{0,1\}^m$ is some string, where $S \oplus s$ is defined as $\{r \oplus s \mid r \in S\}$. Note that for every such shift s, the cardinality of $S \oplus s$ is the same as the cardinality of S. Hence, if $F(x) = 0$, and so S is "tiny", then for every polynomial number of shifts $s_0, \ldots, s_k \in \{0,1\}^m$, the union of the sets $S \oplus s_i$ will not cover $\{0,1\}^m$. On the other hand, we will show that if S is very large, then there exists a polynomial number of such shifts such as $\cup_{i=0}^{k-1}(S \oplus s_i) = \{0,1\}^m$.

We can express the condition that there exists $s_0, \ldots, s_{k-1}$ such that $\cup_{i\in[k]}(S \oplus s_i) = \{0,1\}^m$ as a statement with a constant number of quantifiers. (Specifically, this condition holds if for *every* $y \in \{0,1\}^m$, there *exists* $s \in S$ and $i \in \{0, \ldots, k-1\}$ such that $y = s \oplus s_i$.)

⋆

Proof of Theorem 20.11. Let $F \in \mathbf{BPP}$. Using Theorem 20.5, there exists a polynomial-time algorithm A such that for every $x \in \{0,1\}^n$,

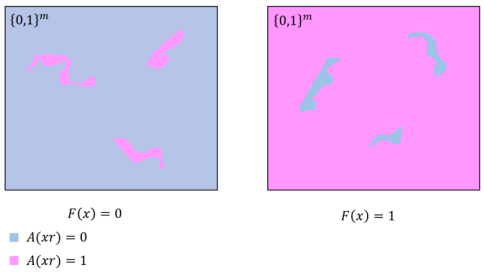


Figure 20.7: If $F \in \mathbf{BPP}$ then through amplification we can ensure that there is an algorithm A to compute F on n-length inputs and using m coins such that $\Pr_{r\sim\{0,1\}^m}[A(xr) \neq F(x)] \ll 1/poly(m)$. Hence if $F(x) = 1$ then almost all of the 2^m choices for r will cause $A(xr)$ to output 1, while if $F(x) = 0$ then $A(xr) = 0$ for almost all r's. To prove the Sipser–Gács Theorem we consider several "shifts" of the set $S \subseteq \{0,1\}^m$ of the coins r such that $A(xr) = 1$. If $F(x) = 1$ then we can find a set of k shifts $s_0, \ldots, s_{k-1}$ for which $\cup_{i\in[k]}(S \oplus s_i) = \{0,1\}^m$. If $F(x) = 0$ then for every such set $|\cup_{i\in[k]} S_i| \leq k|S| \ll 2^m$. We can phrase the question of whether there is such a set of shifts using a constant number of quantifiers, and so can solve it in polynomial time if $\mathbf{P} = \mathbf{NP}$.

$\Pr_{r\in\{0,1\}^m}[A(xr) = F(x)] \geq 1 - 2^{-n}$ where m is polynomial in n. In particular (since an exponential dominates a polynomial, and we can always assume n is sufficiently large), it holds that

$$\Pr_{r\in\{0,1\}^m}[A(xr) = F(x)] \geq 1 - \tfrac{1}{10m^2} \; . \tag{20.8}$$

Let $x \in \{0,1\}^n$, and let $S_x \subseteq \{0,1\}^m$ be the set $\{r \in \{0,1\}^m \; : A(xr) = 1\}$. By our assumption, if $F(x) = 0$ then $|S_x| \leq \frac{1}{10m^2}2^m$ and if $F(x) = 1$ then $|S_x| \geq (1 - \frac{1}{10m^2})2^m$.

For a set $S \subseteq \{0,1\}^m$ and a string $s \in \{0,1\}^m$, we define the set $S \oplus s$ to be $\{r \oplus s \; : \; r \in S\}$ where $\oplus$ denotes the XOR operation. That is, $S \oplus s$ is the set S "shifted" by s. Note that $|S \oplus s| = |S|$. (Please make sure that you see why this is true.)

The heart of the proof is the following two claims:

CLAIM I: For every subset $S \subseteq \{0,1\}^m$, if $|S| \leq \frac{1}{1000m}2^m$, then for every $s_0, \ldots, s_{100m-1} \in \{0,1\}^m$, $\cup_{i\in[100m]}(S \oplus s_i) \subsetneq \{0,1\}^m$.

CLAIM II: For every subset $S \subseteq \{0,1\}^m$, if $|S| \geq \frac{1}{2}2^m$ then there exist a set of string $s_0, \ldots, s_{100m-1}$ such that $\cup_{i\in[100m]}(S \oplus s_i) = \{0,1\}^m$.

CLAIM I and CLAIM II together imply the theorem. Indeed, they mean that under our assumptions, for every $x \in \{0,1\}^n$, $F(x) = 1$ if and only if

$$\exists_{s_0,\ldots,s_{100m-1}\in\{0,1\}^m} \cup_{i\in[100m]} (S_x \oplus s_i) = \{0,1\}^m$$

which we can re-write as

$$\exists_{s_0,\ldots,s_{100m-1}\in\{0,1\}^m} \forall_{w\in\{0,1\}^m} \Big(w \in (S_x \oplus s_0) \vee w \in (S_x \oplus s_1) \vee \cdots w \in (S_x \oplus s_{100m-1}) \Big)$$

or equivalently

$$\exists_{s_0,\ldots,s_{100m-1}\in\{0,1\}^m} \forall_{w\in\{0,1\}^m} \Big(A(x(w \oplus s_0)) = 1 \vee A(x(w \oplus s_1)) = 1 \vee \cdots \vee A(x(w \oplus s_{100m-1})) = 1 \Big)$$

which (since A is computable in polynomial time) is exactly the type of statement shown in Theorem 16.6 to be decidable in polynomial time if **P** = **NP**.

We see that all that is left is to prove **CLAIM I** and **CLAIM II**. **CLAIM I** follows immediately from the fact that

$$\left|\cup_{i\in[100m-1]} S_x \oplus s_i\right| \leq \sum_{i=0}^{100m-1} |S_x \oplus s_i| = \sum_{i=0}^{100m-1} |S_x| = 100m|S_x| \; .$$

To prove **CLAIM II**, we will use a technique known as the *probabilistic method* (see the proof of Lemma 20.12 for a more extensive discussion). Note that this is a completely different use of probability

than in the theorem statement, we just use the methods of probability to prove an *existential* statement.

Proof of CLAIM II: Let $S \subseteq \{0,1\}^m$ with $|S| \geq 0.5 \cdot 2^m$ be as in the claim's statement. Consider the following probabilistic experiment: we choose $100m$ random shifts $s_0, \ldots, s_{100m-1}$ independently at random in $\{0,1\}^m$, and consider the event *GOOD* that $\cup_{i \in [100m]} (S \oplus s_i) = \{0,1\}^m$. To prove CLAIM II it is enough to show that $\Pr[GOOD] > 0$, since that means that in particular there must *exist* shifts $s_0, \ldots, s_{100m-1}$ that satisfy this condition.

For every $z \in \{0,1\}^m$, define the event BAD_z to hold if $z \notin \cup_{i \in [100m-1]} (S \oplus s_i)$. The event *GOOD* holds if BAD_z fails for every $z \in \{0,1\}^m$, and so our goal is to prove that $\Pr[\cup_{z \in \{0,1\}^m} BAD_z] < 1$. By the union bound, to show this, it is enough to show that $\Pr[BAD_z] < 2^{-m}$ for every $z \in \{0,1\}^m$. Define the event BAD_z^i to hold if $z \notin S \oplus s_i$. Since every shift s_i is chosen independently, for every fixed z the events $BAD_z^0, \ldots, BAD_z^{100m-1}$ are mutually independent,[4] and hence

$$\Pr[BAD_z] = \Pr[\cap_{i \in [100m-1]} BAD_z^i] = \prod_{i=0}^{100m-1} \Pr[BAD_z^i] \,. \tag{20.9}$$

So this means that the result will follow by showing that $\Pr[BAD_z^i] \leq \frac{1}{2}$ for every $z \in \{0,1\}^m$ and $i \in [100m]$ (as that would allow to bound the right-hand side of (20.9) by 2^{-100m}). In other words, we need to show that for every $z \in \{0,1\}^m$ and set $S \subseteq \{0,1\}^m$ with $|S| \geq \frac{1}{2} 2^m$,

$$\Pr_{s \sim \{0,1\}^m} [z \in S \oplus s] \geq \tfrac{1}{2} \,. \tag{20.10}$$

To show this, we observe that $z \in S \oplus s$ if and only if $s \in S \oplus z$ (can you see why). Hence we can rewrite the probability on the left-hand side of (20.10) as $\Pr_{s \sim \{0,1\}^m}[s \in S \oplus z]$ which simply equals $|S \oplus z|/2^m = |S|/2^m \geq 1/2$! This concludes the proof of **CLAIM I** and hence of Theorem 20.11.

■

[4] The condition of independence here is subtle. It is *not* the case that all of the $2^m \times 100m$ events $\{BAD_z^i\}_{z \in \{0,1\}^m, i \in [100m]}$ are mutually independent. Only for a fixed $z \in \{0,1\}^m$, the $100m$ events of the form BAD_z^i are mutually independent.

20.6 NON-CONSTRUCTIVE EXISTENCE OF PSEUDORANDOM GENERATORS (ADVANCED, OPTIONAL)

We now show that, if we don't insist on *constructivity* of pseudorandom generators, then we can show that there exist pseudorandom generators with output that is *exponentially larger* in the input length.

Lemma 20.12 — Existence of inefficient pseudorandom generators. There is some absolute constant C such that for every ϵ, T, if $\ell > C(\log T + \log(1/\epsilon))$ and $m \leq T$, then there is a (T, ϵ) pseudorandom generator $G : \{0,1\}^\ell \to \{0,1\}^m$.

Proof Idea:

The proof uses an extremely useful technique known as the "probabilistic method" which is not too hard mathematically but can be confusing at first.[5] The idea is to give a "non-constructive" proof of existence of the pseudorandom generator G by showing that if G was chosen at random, then the probability that it would be a valid (T, ϵ) pseudorandom generator is positive. In particular this means that there *exists* a single G that is a valid (T, ϵ) pseudorandom generator. The probabilistic method is just a *proof technique* to demonstrate the existence of such a function. Ultimately, our goal is to show the existence of a *deterministic* function G that satisfies the condition.

[5] There is a whole (highly recommended) book by Alon and Spencer devoted to this method.

⋆

The above discussion might be rather abstract at this point, but would become clearer after seeing the proof.

Proof of Lemma 20.12. Let ϵ, T, ℓ, m be as in the lemma's statement. We need to show that there exists a function $G : \{0,1\}^\ell \to \{0,1\}^m$ that "fools" every T line program P in the sense of (20.7). We will show that this follows from the following claim:

Claim I: For every fixed NAND-CIRC program P, if we pick $G : \{0,1\}^\ell \to \{0,1\}^m$ *at random* then the probability that (20.7) is violated is at most 2^{-T^2}.

Before proving Claim I, let us see why it implies Lemma 20.12. We can identify a function $G : \{0,1\}^\ell \to \{0,1\}^m$ with its "truth table" or simply the list of evaluations on all its possible 2^ℓ inputs. Since each output is an m bit string, we can also think of G as a string in $\{0,1\}^{m \cdot 2^\ell}$. We define $\mathcal{F}_\ell^m$ to be the set of all functions from $\{0,1\}^\ell$ to $\{0,1\}^m$. As discussed above we can identify $\mathcal{F}_\ell^m$ with $\{0,1\}^{m \cdot 2^\ell}$ and choosing a random function $G \sim \mathcal{F}_\ell^m$ corresponds to choosing a random $m \cdot 2^\ell$-long bit string.

For every NAND-CIRC program P let B_P be the event that, if we choose G at random from $\mathcal{F}_\ell^m$ then (20.7) is violated with respect to the program P. It is important to understand what is the sample space that the event B_P is defined over, namely this event depends on the choice of G and so B_P is a subset of $\mathcal{F}_\ell^m$. An equivalent way to define the event B_P is that it is the subset of all functions mapping $\{0,1\}^\ell$ to $\{0,1\}^m$ that violate (20.7), or in other words:

$$B_P = \left\{ G \in \mathcal{F}_\ell^m \;\middle|\; \left| \tfrac{1}{2^\ell} \sum_{s \in \{0,1\}^\ell} P(G(s)) - \tfrac{1}{2^m} \sum_{r \in \{0,1\}^m} P(r) \right| > \epsilon \right\} \tag{20.11}$$

(We've replaced here the probability statements in (20.7) with the equivalent sums so as to reduce confusion as to what is the sample space that B_P is defined over.)

To understand this proof it is crucial that you pause here and see how the definition of B_P above corresponds to (20.11). This may well take re-reading the above text once or twice, but it is a good exercise at parsing probabilistic statements and learning how to identify the *sample space* that these statements correspond to.

Now, we've shown in Theorem 5.2 that up to renaming variables (which makes no difference to program's functionality) there are $2^{O(T \log T)}$ NAND-CIRC programs of at most T lines. Since $T \log T < T^2$ for sufficiently large T, this means that if Claim I is true, then by the union bound it holds that the probability of the union of B_P over *all* NAND-CIRC programs of at most T lines is at most $2^{O(T \log T)} 2^{-T^2} < 0.1$ for sufficiently large T. What is important for us about the number 0.1 is that it is smaller than 1. In particular this means that there *exists* a single $G^* \in \mathcal{F}_\ell^m$ such that G^* *does not* violate (20.7) with respect to any NAND-CIRC program of at most T lines, but that precisely means that G^* is a (T, ϵ) pseudorandom generator.

Hence to conclude the proof of Lemma 20.12, it suffices to prove Claim I. Choosing a random $G : \{0,1\}^\ell \to \{0,1\}^m$ amounts to choosing $L = 2^\ell$ random strings $y_0, \ldots, y_{L-1} \in \{0,1\}^m$ and letting $G(x) = y_x$ (identifying $\{0,1\}^\ell$ and $[L]$ via the binary representation). This means that proving the claim amounts to showing that for every fixed function $P : \{0,1\}^m \to \{0,1\}$, if $L > 2^{C(\log T + \log \epsilon)}$ (which by setting $C > 4$, we can ensure is larger than $10T^2/\epsilon^2$) then the probability that

$$\left| \frac{1}{L} \sum_{i=0}^{L-1} P(y_i) - \Pr_{s \sim \{0,1\}^m}[P(s) = 1] \right| > \epsilon \tag{20.12}$$

is at most 2^{-T^2}.

(20.12) follows directly from the Chernoff bound. Indeed, if we let for every $i \in [L]$ the random variable X_i denote $P(y_i)$, then since $y_0, \ldots, y_{L-1}$ is chosen independently at random, these are independently and identically distributed random variables with mean $\mathbb{E}_{y \sim \{0,1\}^m}[P(y)] = \Pr_{y \sim \{0,1\}^m}[P(y) = 1]$ and hence the probability that they deviate from their expectation by ϵ is at most $2 \cdot 2^{-\epsilon^2 L/2}$.

■

✓ **Chapter Recap**

- We can model randomized algorithms by either adding a special "coin toss" operation or assuming an extra randomly chosen input.

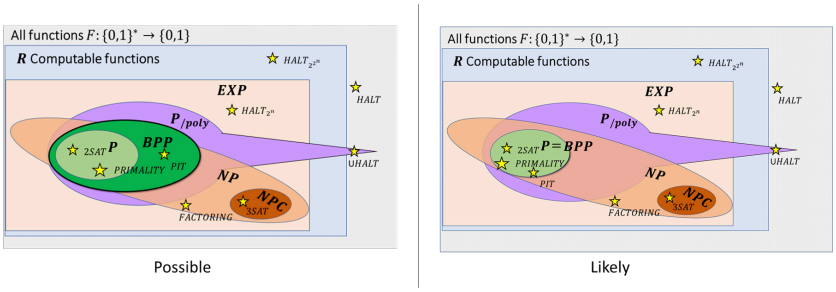


Figure 20.8: The relation between **BPP** and the other complexity classes that we have seen. We know that $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{EXP}$ and $\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$ but we don't know how **BPP** compares with **NP** and can't rule out even $\mathbf{BPP} = \mathbf{EXP}$. Most evidence points out to the possibliity that $\mathbf{BPP} = \mathbf{P}$.

- The class **BPP** contains the set of Boolean functions that can be computed by polynomial time randomized algorithms.
- **BPP** is a *worst case* class of computation: a randomized algorithm to compute a function must compute it correctly with high probability *on every input*.
- We can *amplify* the success probability of randomized algorithm from any value strictly larger than $1/2$ into a success probability that is *exponentially close to* 1.
- We know that $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{EXP}$.
- We also know that $\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$.
- The relation between **BPP** and **NP** is not known, but we do know that if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{BPP} = \mathbf{P}$.
- Pseudorandom generators are objects that take a short random "seed" and expand it to a much longer output that "appears random" for efficient algorithms. We conjecture that exponentially strong pseudorandom generators exist. Under this conjecture, $\mathbf{BPP} = \mathbf{P}$.

20.7 EXERCISES

20.8 BIBLIOGRAPHICAL NOTES

In this chapter we ignore the issue of how we actually get random bits in practice. The output of many physical processes, whether it is thermal heat, network and hard drive latency, user typing pattern and mouse movements, and more can be thought of as a binary string sampled from some distribution μ that might have significant unpredictability (or *entropy*) but is not necessarily the *uniform* distribution over $\{0,1\}^n$. Indeed, as this paper shows, even (real-world) coin tosses do not have exactly the distribution of a uniformly random string. Therefore, to use the resulting measurements for randomized algorithms, one typically needs to apply a "distillation" or random-

ness extraction process to the raw measurements to transform them to the uniform distribution. Vadhan's book [Vad+12] is an excellent source for more discussion on both randomness extractors and pseudorandom generators.

The name **BPP** stands for "bounded probability polynomial time". This is an historical accident: this class probably should have been called **RP** or **PP** but both names were taken by other classes.

The proof of Theorem 20.8 actually yields more than its statement. We can use the same "unrolling the loop" arguments we've used before to show that the restriction to $\{0,1\}^n$ of every function in **BPP** is also computable by a polynomial-size RNAND-CIRC program (i.e., NAND-CIRC program with the `RAND` operation). Like in the **P** vs $SIZE(poly(n))$ case, there are also functions outside **BPP** whose restrictions can be computed by polynomial-size RNAND-CIRC programs. Nevertheless the proof of Theorem 20.8 shows that even such functions can be computed by polynomial-sized NAND-CIRC programs without using the `rand` operations. This can be phrased as saying that $BPSIZE(T(n)) \subseteq SIZE(O(nT(n)))$ (where $BPSIZE$ is defined in the natural way using RNAND progams). The stronger version of Theorem 20.8 we mentioned can be phrased as saying that $\mathbf{BPP}_{/\mathbf{poly}} = \mathbf{P}_{/\mathbf{poly}}$.

V

ADVANCED TOPICS

Learning Objectives:

- Definition of perfect secrecy
- The one-time pad encryption scheme
- Necessity of long keys for perfect secrecy
- Computational secrecy and the derandomized one-time pad.
- Public key encryption
- A taste of advanced topics

21
Cryptography

> *"Human ingenuity cannot concoct a cipher which human ingenuity cannot resolve."*, Edgar Allen Poe, 1841

> *"A good disguise should not reveal the person's height"*, Shafi Goldwasser and Silvio Micali, 1982

> *""Perfect Secrecy" is defined by requiring of a system that after a cryptogram is intercepted by the enemy the a posteriori probabilities of this cryptogram representing various messages be identically the same as the a priori probabilities of the same messages before the interception. It is shown that perfect secrecy is possible but requires, if the number of messages is finite, the same number of possible keys."*, Claude Shannon, 1945

> *"We stand today on the brink of a revolution in cryptography."*, Whitfeld Diffie and Martin Hellman, 1976

Cryptography - the art or science of "secret writing" - has been around for several millennia, and for almost all of that time Edgar Allan Poe's quote above held true. Indeed, the history of cryptography is littered with the figurative corpses of cryptosystems believed secure and then broken, and sometimes with the actual corpses of those who have mistakenly placed their faith in these cryptosystems.

Yet, something changed in the last few decades, which is the "revolution" alluded to (and to a large extent initiated by) Diffie and Hellman's 1976 paper quoted above. New cryptosystems have been found that have not been broken despite being subjected to immense efforts involving both human ingenuity and computational power on a scale that completely dwarves the "code breakers" of Poe's time. Even more amazingly, these cryptosystems are not only seemingly unbreakable, but they also achieve this under much harsher conditions. Not only do today's attackers have more computational power, but they also have more data to work with. In Poe's age, an attacker would be lucky if they got access to more than a few encryptions of known messages. These days attackers might have massive amounts of data—terabytes

or more—at their disposal. In fact, with *public key* encryption, an attacker can generate as many ciphertexts as they wish.

The key to this success has been a clearer understanding of both how to *define* security for cryptographic tools and how to relate this security to *concrete computational problems*. Cryptography is a vast and continuously changing topic, but we will touch on some of these issues in this chapter.

This chapter: A non-mathy overview

Cryptography cannot be covered in a single chapter, and so this chapter merely gives a "taste" of crypto, focusing on the aspects most related to computational complexity. For a more extensive treatment, see my lecture notes from which this chapter is adapted. We will discuss some "classical cryptosystems" and show how we can *mathematically define* security of encryption, and use the *one-time pad* to achieve an encryption that provably satisfies this definition. We will then see the fundamental limitation of this definition, and how to bypass it we need to relax security by only restricting attention to attackers that have *bounded computational resources*. This notion of *computational security* is inherently tied to computational complexity and the **P** vs **NP** question. We will also give a small taste of some of the "paradoxical" cryptographic constructions that go way beyond encryption, including public-key cryptography, fully-homomorphic encryption, and multi-party secure computation.

21.1 CLASSICAL CRYPTOSYSTEMS

A great many cryptosystems have been devised and broken throughout the ages. Let us recount just some of these stories. In 1587, Mary, Queen of Scots, and the heir to the throne of England, wanted to arrange the assassination of her cousin, Queen Elizabeth I of England, so that she could ascend to the throne and finally escape the house arrest under which she had been for the last 18 years. As part of this complicated plot, she sent a coded letter to Sir Anthony Babington.

Mary used what's known as a *substitution cipher* where each letter is transformed into a different obscure symbol (see Fig. 21.1). At a first look, such a letter might seem rather inscrutable—a meaningless sequence of strange symbols. However, after some thought, one might recognize that these symbols *repeat* several times and moreover that different symbols repeat with different frequencies. Now it doesn't take a large leap of faith to assume that perhaps each symbol corresponds to a different letter and the more frequent symbols correspond

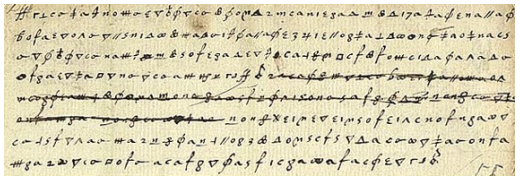


Figure 21.1: Snippet from encrypted communication between queen Mary and Sir Babington

to letters that occur in the alphabet with higher frequency. From this observation, there is a short gap to completely breaking the cipher, which was in fact done by Queen Elizabeth's spies, who used the decoded letters to learn of all the co-conspirators and to convict Queen Mary of treason, a crime for which she was executed. Trusting in superficial security measures (such as using "inscrutable" symbols) is a trap that users of cryptography have been falling into again and again over the years. (As with many things, this is the subject of a great XKCD cartoon, see Fig. 21.2.)

The Vigenère cipher is named after Blaise de Vigenère, who described it in a book in 1586 (though it was invented earlier by Bellaso). The idea is to use a collection of substitution cyphers: if there are n different ciphers then the first letter of the plaintext is encoded with the first cipher, the second with the second cipher, the nth with the nth cipher, and then the $n+1^{st}$ letter is again encoded with the first cipher. The key k is usually a word or a phrase of n letters. The ith substitution cipher will shift each letter by the same shift needed to get from A to k_i. If k_i is C, for example, the ith substitution cipher will shift every letter by two places. This "flattens" the frequencies and makes it much harder to do frequency analysis, which is why this cipher was considered "unbreakable" for 300+ years and got the nickname "le chiffre indéchiffrable" ("the unbreakable cipher"). Nevertheless, Charles Babbage cracked the Vigenère cipher in 1854 (though he did not publish it). In 1863 Friedrich Kasiski broke the cipher and published the result. The idea is that once you guess the length of the cipher, you can reduce the task to breaking a simple substitution cipher which can be done via frequency analysis (can you see why?). Confederate generals used Vigenère regularly during the civil war, and their messages were routinely cryptanalzed by Union officers.

The *Enigma* cipher was a mechanical cipher (looking like a typewriter, see Fig. 21.5) where each letter typed would get mapped into a different letter depending on the (rather complicated) key and current state of the machine,d which had several rotors that rotated at different paces. An identically wired machine at the other end could be used to decrypt. Just as many ciphers in history, this has also been believed by the Germans to be "impossible to break" and even quite late in the war they refused to believe it was broken despite mounting evidence to that effect. (In fact, some German generals refused to believe it was broken even *after* the war.) Breaking Enigma was an heroic effort which was initiated by the Poles and then completed by the British at Bletchley Park, with Alan Turing (of the Turing machine) playing a key role. As part of this effort the Brits built arguably the world's first large scale mechanical computation devices (though they looked more similar to washing machines than to iPhones). They were

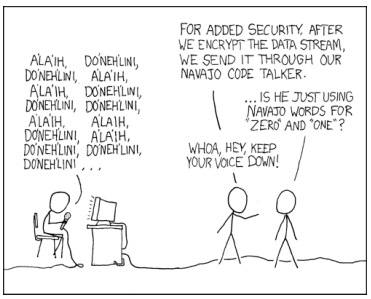


Figure 21.2: XKCD's take on the added security of using uncommon symbols

Figure 21.3: Confederate Cipher Disk for implementing the Vigenère cipher

Figure 21.4: Confederate encryption of the message "Gen'l Pemberton: You can expect no help from this side of the river. Let Gen'l Johnston know, if possible, when you can attack the same point on the enemy's lines. Inform me also and I will endeavor to make a diversion. I have sent some caps. I subjoin a despatch from General Johnston."

also helped along the way by some quirks and errors of the German operators. For example, the fact that their messages ended with "Heil Hitler" turned out to be quite useful.

Here is one entertaining anecdote: the Enigma machine would never map a letter to itself. In March 1941, Mavis Batey, a cryptanalyst at Bletchley Park received a very long message that she tried to decrypt. She then noticed a curious property— the message did *not* contain the letter "L".[1] She realized that the probability that no "L"'s appeared in the message is too small for this to happen by chance. Hence she surmised that the original message must have been composed *only* of L's. That is, it must have been the case that the operator, perhaps to test the machine, have simply sent out a message where he repeatedly pressed the letter "L". This observation helped her decode the next message, which helped inform of a planned Italian attack and secure a resounding British victory in what became known as "the Battle of Cape Matapan". Mavis also helped break another Enigma machine. Using the information she provided, the Brits were able to feed the Germans with the false information that the main allied invasion would take place in Pas de Calais rather than on Normandy.

In the words of General Eisenhower, the intelligence from Bletchley Park was of "priceless value". It made a huge difference for the Allied war effort, thereby shortening World War II and saving millions of lives. See also this interview with Sir Harry Hinsley.

Figure 21.5: In the *Enigma* mechanical cipher the secret key would be the settings of the rotors and internal wires. As the operator typed up their message, the encrypted version appeared in the display area above, and the internal state of the cipher was updated (and so typing the same letter twice would generally result in two different letters output). Decrypting follows the same process: if the sender and receiver are using the same key then typing the ciphertext would result in the plaintext appearing in the display.

[1] Here is a nice exercise: compute (up to an order of magnitude) the probability that a 50-letter long message composed of random letters will end up not containing the letter "L".

21.2 DEFINING ENCRYPTION

Many of the troubles that cryptosystem designers faced over history (and still face!) can be attributed to not properly defining or understanding the goals they want to achieve in the first place. Let us focus on the setting of *private key encryption*. (This is also known as "symmetric encryption"; for thousands of years, "private key encryption" was synonymous with encryption and only in the 1970s was the concept of *public key encryption* invented, see Definition 21.11.) A *sender* (traditionally called "Alice") wants to send a message (known also as a *plaintext*) $x \in \{0,1\}^*$ to a *receiver* (traditionally called "Bob"). They would like their message to be kept secret from an *adversary* who listens in or "eavesdrops" on the communication channel (and is traditionally called "Eve").

Alice and Bob share a *secret key* $k \in \{0,1\}^*$. (While the letter k is often used elsewhere in the book to denote a natural number, in this chapter we use it to denote the string corresponding to a secret key.) Alice uses the key k to "scramble" or *encrypt* the plaintext x into a *ciphertext* y, and Bob uses the key k to "unscramble" or *decrypt* the ciphertext y back into the plaintext x. This motivates the following definition which attempts to capture what it means for an encryption

scheme to be *valid* or "make sense", regardless of whether or not it is *secure*:

Definition 21.1 — Valid encryption scheme. Let $L : \mathbb{N} \to \mathbb{N}$ and $C : \mathbb{N} \to \mathbb{N}$ be two functions mapping natural numbers to natural numbers. A pair of polynomial-time computable functions (E, D) mapping strings to strings is a *valid private key encryption scheme* (or *encryption scheme* for short) with plaintext length function $L(\cdot)$ and ciphertext length function $C(\cdot)$ if for every $n \in \mathbb{N}$, $k \in \{0,1\}^n$ and $x \in \{0,1\}^{L(n)}$, $|E_k(x)| = C(n)$ and

$$D(k, E(k, x)) = x \; . \tag{21.1}$$

We will often write the first input (i.e., the key) to the encryption and decryption as a subscript and so can write (21.1) also as $D_k(E_k(x)) = x$.

Solved Exercise 21.1 — Lengths of ciphertext and plaintext. Prove that for every valid encryption scheme (E, D) with functions L, C. $C(n) \geq L(n)$ for every n.

Solution:

For every fixed key $k \in \{0,1\}^n$, the equation (21.1) implies that the map $y \mapsto D_k(y)$ inverts the map $x \mapsto E_k(x)$, which in particular means that the map $x \mapsto E_k(x)$ must be one to one. Hence its codomain must be at least as large as its domain, and since its domain is $\{0,1\}^{L(n)}$ and its codomain is $\{0,1\}^{C(n)}$ it follows that $C(n) \geq L(n)$.

Since the ciphertext length is always at least the plaintext length (and in most applications it is not much longer than that), we typically focus on the plaintext length as the quantity to optimize in an encryption scheme. The *larger* $L(n)$ is, the better the scheme, since it means we need a shorter secret key to protect messages of the same length.

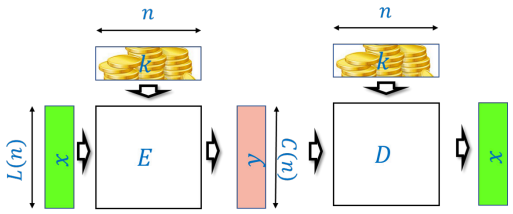


Figure 21.6: A private-key encryption scheme is a pair of algorithms E, D such that for every key $k \in \{0,1\}^n$ and plaintext $x \in \{0,1\}^{L(n)}$, $y = E_k(x)$ is a ciphertext of length $C(n)$. The encryption scheme is *valid* if for every such y, $D_k(y) = x$. That is, the decryption of an encryption of x is x, as long as both encryption and decryption use the same key.

21.3 DEFINING SECURITY OF ENCRYPTION

Definition 21.1 says nothing about the *security* of E and D, and even allows the trivial encryption scheme that ignores the key altogether and sets $E_k(x) = x$ for every x. Defining security is not a trivial matter.

P

You would appreciate the subtleties of defining security of encryption more if at this point you take a five minute break from reading, and try (possibly with a partner) to brainstorm on how you would mathematically define the notion that an encryption scheme is *secure*, in the sense that it protects the secrecy of the plaintext x.

Throughout history, many attacks on cryptosystems were rooted in the cryptosystem designers' reliance on "security through obscurity"— trusting that the fact their *methods* are not known to their enemy will protect them from being broken. This is a faulty assumption - if you reuse a method again and again (even with a different key each time) then eventually your adversaries will figure out what you are doing. And if Alice and Bob meet frequently in a secure location to decide on a new method, they might as well take the opportunity to exchange their secrets. These considerations led Auguste Kerckhoffs in 1883 to state the following principle:

A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.[2]

[2] The actual quote is "Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi" loosely translated as "The system must not require secrecy and can be stolen by the enemy without causing trouble". According to Steve Bellovin the NSA version is "assume that the first copy of any device we make is shipped to the Kremlin".

Why is it OK to assume the key is secret and not the algorithm? Because we can always choose a fresh key. But of course that won't help us much if our key is "1234" or "passw0rd!". In fact, if you use *any* deterministic algorithm to choose the key then eventually your adversary will figure this out. Therefore for security we must choose the key at *random* and can restate Kerckhoffs's principle as follows:

There is no secrecy without randomness

This is such a crucial point that is worth repeating:

Big Idea 26 There is no *secrecy* without *randomness*.

At the heart of every cryptographic scheme there is a secret key, and the secret key is always chosen at random. A corollary of that is that to understand cryptography, you need to know probability theory.

R

Remark 21.2 — Randomness in the real world. Choosing the secrets for cryptography requires generating randomness, which is often done by measuring some "unpredictable" or "high entropy" data, and then applying hash functions to the result to "extract" a

uniformly random string. Great care must be taken in doing this, and randomness generators often turn out to be the Achilles heel of secure systems.

In 2006 a programmer removed a line of code from the procedure to generate entropy in OpenSSL package distributed by Debian since it caused a warning in some automatic verification code. As a result for two years (until this was discovered) all the randomness generated by this procedure used only the process ID as an "unpredictable" source. This means that all communication done by users in that period is fairly easily breakable (and in particular, if some entities recorded that communication they could break it also retroactively). See XKCD's take on that incident.

In 2012 two separate teams of researchers scanned a large number of RSA keys on the web and found out that about 4 percent of them are easy to break. The main issue were devices such as routers, internet-connected printers and such. These devices sometimes run variants of Linux—a desktop operating system—but without a hard drive, mouse or keyboard, they don't have access to many of the entropy sources that desktops have. Coupled with some good old fashioned ignorance of cryptography and software bugs, this led to many keys that are downright trivial to break, see this blog post and this web page for more details.

Since randomness is so crucial to security, breaking the procedure to generate randomness can lead to a complete break of the system that uses this randomness. Indeed, the Snowden documents, combined with observations of Shumow and Ferguson, strongly suggest that the NSA has deliberately inserted a *trapdoor* in one of the pseudorandom generators published by the National Institute of Standards and Technology (NIST). Fortunately, this generator wasn't widely adapted, but apparently the NSA did pay 10 million dollars to RSA Security so the latter would make this generator the default option in their products.

21.4 PERFECT SECRECY

If you think about encryption scheme security for a while, you might come up with the following principle for defining security: *"An encryption scheme is secure if it is not possible to recover the key k from $E_k(x)$"*. However, a moment's thought shows that the key is not really what we're trying to protect. After all, the whole point of an encryption is to protect the confidentiality of the *plaintext* x. So, we can try to define that *"an encryption scheme is secure if it is not possible to recover the plaintext x from $E_k(x)$"*. Yet it is not clear what this means either. Suppose that an encryption scheme reveals the first 10 bits of the plaintext

x. It might still not be possible to recover x completely, but on an intuitive level, this seems like it would be extremely unwise to use such an encryption scheme in practice. Indeed, often even *partial information* about the plaintext is enough for the adversary to achieve its goals.

The above thinking led Shannon in 1945 to formalize the notion of *perfect secrecy*, which is that an encryption reveals absolutely nothing about the message. There are several equivalent ways to define it, but perhaps the cleanest one is the following:

Definition 21.3 — Perfect secrecy. A valid encryption scheme (E, D) with plaintext length $L(\cdot)$ is *perfectly secret* if for every $n \in \mathbb{N}$ and plaintexts $x, x' \in \{0,1\}^{L(n)}$, the following two distributions Y and Y' over $\{0,1\}^*$ are identical:

- Y is obtained by sampling $k \sim \{0,1\}^n$ and outputting $E_k(x)$.
- Y' is obtained by sampling $k \sim \{0,1\}^n$ and outputting $E_k(x')$.

P

This definition might take more than one reading to parse. Try to think of how this condition would correspond to your intuitive notion of "learning no information" about x from observing $E_k(x)$, and to Shannon's quote in the beginning of this chapter.

In particular, suppose that you knew ahead of time that Alice sent either an encryption of x or an encryption of x'. Would you learn anything new from observing the encryption of the message that Alice actually sent? It may help you to look at Fig. 21.7.

21.4.1 Example: Perfect secrecy in the battlefield

To understand Definition 21.3, suppose that Alice sends only one of two possible messages: "attack" or "retreat", which we denote by x_0 and x_1 respectively, and that she sends each one of those messages with probability $1/2$. Let us put ourselves in the shoes of *Eve*, the eavesdropping adversary. A priori we would have guessed that Alice sent either x_0 or x_1 with probability $1/2$. Now we observe $y = E_k(x_i)$ where k is a uniformly chosen key in $\{0,1\}^n$. How does this new information cause us to update our beliefs on whether Alice sent the plaintext x_0 or the plaintext x_1?

P

Before reading the next paragraph, you might want to try the analysis yourself. You may find it useful to

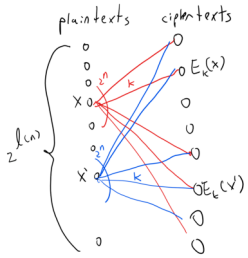


Figure 21.7: For any key length n, we can visualize an encryption scheme (E, D) as a graph with a vertex for every one of the $2^{L(n)}$ possible plaintexts and for every one of the ciphertexts in $\{0,1\}^*$ of the form $E_k(x)$ for $k \in \{0,1\}^n$ and $x \in \{0,1\}^{L(n)}$. For every plaintext x and key k, we add an edge labeled k between x and $E_k(x)$. By the validity condition, if we pick any fixed key k, the map $x \mapsto E_k(x)$ must be one-to-one. The condition of perfect secrecy simply corresponds to requiring that every two plaintexts x and x' have exactly the same set of neighbors (or multi-set, if there are parallel edges).

look at the Wikipedia entry on Bayesian Inference or these MIT lecture notes.

Let us define $p_0(y)$ to be the probability (taken over $k \sim \{0,1\}^n$) that $y = E_k(x_0)$ and similarly $p_1(y)$ to be $\Pr_{k\sim\{0,1\}^n}[y = E_k(x_1)]$. Note that, since Alice chooses the message to send at random, our a priori probability for observing y is $\frac{1}{2}p_0(y) + \frac{1}{2}p_1(y)$. However, as per Definition 21.3, the perfect secrecy condition guarantees that $p_0(y) = p_1(y)$! Let us denote the number $p_0(y) = p_1(y)$ by p. By the formula for conditional probability, the probability that Alice sent the message x_0 conditioned on our observation y is simply

$$\Pr[i = 0 | y = E_k(x_i)] = \frac{\Pr[i = 0 \wedge y = E_k(x_i)]}{\Pr[y = E_k(x)]} . \tag{21.2}$$

(The equation (21.2) is a special case of *Bayes' rule* which, although a simple restatement of the formula for conditional probability, is an extremely important and widely used tool in statistics and data analysis.)

Since the probability that $i = 0$ and y is the ciphertext $E_k(0)$ is equal to $\frac{1}{2} \cdot p_0(y)$, and the a priori probability of observing y is $\frac{1}{2}p_0(y) + \frac{1}{2}p_1(y)$, we can rewrite (21.2) as

$$\Pr[i = 0 | y = E_k(x_i)] = \frac{\frac{1}{2}p_0(y)}{\frac{1}{2}p_0(y) + \frac{1}{2}p_1(y)} = \frac{p}{p+p} = \frac{1}{2}$$

using the fact that $p_0(y) = p_1(y) = p$. This means that observing the ciphertext y did not help us at all! We still would not be able to guess whether Alice sent "attack" or "retreat" with better than 50/50 odds!

This example can be vastly generalized to show that perfect secrecy is indeed "perfect" in the sense that observing a ciphertext gives Eve *no additional information* about the plaintext beyond her a priori knowledge.

21.4.2 Constructing perfectly secret encryption

Perfect secrecy is an extremely strong condition, and implies that an eavesdropper does not learn *any* information from observing the ciphertext. You might think that an encryption scheme satisfying such a strong condition will be impossible, or at least extremely complicated, to achieve. However it turns out we can in fact obtain a perfectly secret encryption scheme fairly easily. Such a scheme for two-bit messages is illustrated in Fig. 21.8.

In fact, this can be generalized to any number of bits:

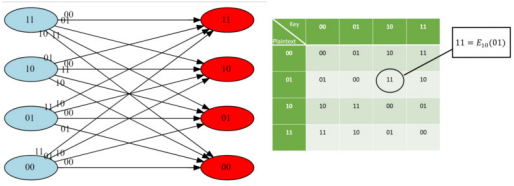


Figure 21.8: A perfectly secret encryption scheme for two-bit keys and messages. The blue vertices represent plaintexts and the red vertices represent ciphertexts, each edge mapping a plaintext x to a ciphertext $y = E_k(x)$ is labeled with the corresponding key k. Since there are four possible keys, the degree of the graph is four and it is in fact a complete bipartite graph. The encryption scheme is valid in the sense that for every $k \in \{0,1\}^2$, the map $x \mapsto E_k(x)$ is one-to-one, which in other words means that the set of edges labeled with k is a *matching*.

Theorem 21.4 — One Time Pad (Vernam 1917, Shannon 1949). There is a perfectly secret valid encryption scheme (E, D) with $L(n) = C(n) = n$.

Proof Idea:

Our scheme is the one-time pad also known as the "Vernam Cipher", see Fig. 21.9. The encryption is exceedingly simple: to encrypt a message $x \in \{0,1\}^n$ with a key $k \in \{0,1\}^n$ we simply output $x \oplus k$ where $\oplus$ is the bitwise XOR operation that outputs the string corresponding to XORing each coordinate of x and k.

⋆

Proof of Theorem 21.4. For two binary strings a and b of the same length n, we define $a \oplus b$ to be the string $c \in \{0,1\}^n$ such that $c_i = a_i + b_i \mod 2$ for every $i \in [n]$. The encryption scheme (E, D) is defined as follows: $E_k(x) = x \oplus k$ and $D_k(y) = y \oplus k$. By the associative law of addition (which works also modulo two), $D_k(E_k(x)) = (x \oplus k) \oplus k = x \oplus (k \oplus k) = x \oplus 0^n = x$, using the fact that for every bit $\sigma \in \{0,1\}$, $\sigma + \sigma \mod 2 = 0$ and $\sigma + 0 = \sigma \mod 2$. Hence (E, D) form a valid encryption.

To analyze the perfect secrecy property, we claim that for every $x \in \{0,1\}^n$, the distribution $Y_x = E_k(x)$ where $k \sim \{0,1\}^n$ is simply the uniform distribution over $\{0,1\}^n$, and hence in particular the distributions Y_x and $Y_{x'}$ are identical for every $x, x' \in \{0,1\}^n$. Indeed, for every particular $y \in \{0,1\}^n$, the value y is output by Y_x if and only if $y = x \oplus k$ which holds if and only if $k = x \oplus y$. Since k is chosen uniformly at random in $\{0,1\}^n$, the probability that k happens to equal $x \oplus y$ is exactly 2^{-n}, which means that every string y is output by Y_x with probability 2^{-n}.

■

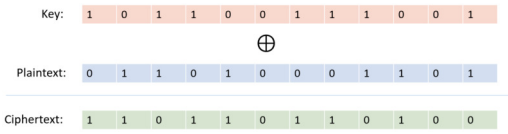


Figure 21.9: In the *one time pad* encryption scheme we encrypt a plaintext $x \in \{0,1\}^n$ with a key $k \in \{0,1\}^n$ by the ciphertext $x \oplus k$ where $\oplus$ denotes the bitwise XOR operation.

P

The argument above is quite simple but is worth reading again. To understand why the one-time pad is perfectly secret, it is useful to envision it as a bipartite graph as we've done in Fig. 21.8. (In fact the encryption scheme of Fig. 21.8 is precisely the one-time pad for $n = 2$.) For every n, the one-time pad encryption scheme corresponds to a bipartite graph with 2^n vertices on the "left side" corresponding to the plaintexts in $\{0,1\}^n$ and 2^n vertices on the "right side" corresponding to the ciphertexts $\{0,1\}^n$. For every $x \in \{0,1\}^n$ and $k \in \{0,1\}^n$, we connect x to the vertex $y = E_k(x)$ with an edge that we label with k. One can see that this is the complete bipartite graph, where every vertex on the left is connected to *all* vertices on the right. In particular this means that for every left vertex x, the distribution on the ciphertexts obtained

by taking a random $k \in \{0,1\}^n$ and going to the neighbor of x on the edge labeled k is the uniform distribution over $\{0,1\}^n$. This ensures the perfect secrecy condition.

21.5 NECESSITY OF LONG KEYS

So, does Theorem 21.4 give the final word on cryptography, and means that we can all communicate with perfect secrecy and live happily ever after? No it doesn't. While the one-time pad is efficient, and gives perfect secrecy, it has one glaring disadvantage: to communicate n bits you need to store a key of length n. In contrast, practically used cryptosystems such as AES-128 have a short key of 128 bits (i.e., 16 bytes) that can be used to protect terabytes or more of communication! Imagine that we all needed to use the one time pad. If that was the case, then if you had to communicate with m people, you would have to maintain (securely!) m huge files that are each as long as the length of the maximum total communication you expect with that person. Imagine that every time you opened an account with Amazon, Google, or any other service, they would need to send you in the mail (ideally with a secure courier) a DVD full of random numbers, and every time you suspected a virus, you'd need to ask all these services for a fresh DVD. This doesn't sound so appealing.

This is not just a theoretical issue. The Soviets have used the one-time pad for their confidential communication since before the 1940's. In fact, even before Shannon's work, the U.S. intelligence already knew in 1941 that the one-time pad is in principle "unbreakable" (see page 32 in the Venona document). However, it turned out that the hassle of manufacturing so many keys for all the communication took its toll on the Soviets and they ended up reusing the same keys for more than one message. They did try to use them for completely different receivers in the (false) hope that this wouldn't be detected. The Venona Project of the U.S. Army was founded in February 1943 by Gene Grabeel (see Fig. 21.10), a former home economics teacher from Madison Heights, Virgnia and Lt. Leonard Zubko. In October 1943, they had their breakthrough when it was discovered that the Russians were reusing their keys. In the 37 years of its existence, the project has resulted in a treasure chest of intelligence, exposing hundreds of KGB agents and Russian spies in the U.S. and other countries, including Julius Rosenberg, Harry Gold, Klaus Fuchs, Alger Hiss, Harry Dexter White and many others.

Unfortunately it turns out that such long keys are *necessary* for perfect secrecy:

Figure 21.10: Gene Grabeel, who founded the U.S. Russian SigInt program on 1 Feb 1943. Photo taken in 1942, see Page 7 in the Venona historical study.

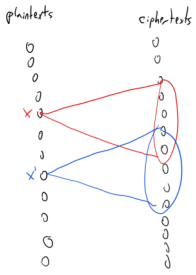


Figure 21.11: An encryption scheme where the number of keys is smaller than the number of plaintexts corresponds to a bipartite graph where the degree is smaller than the number of vertices on the left side. Together with the validity condition this implies that there will be two left vertices x, x' with non-identical neighborhoods, and hence the scheme does *not* satisfy perfect secrecy.

Theorem 21.5 — Perfect secrecy requires long keys. For every perfectly secret encryption scheme (E, D) the length function L satisfies $L(n) \leq n$.

Proof Idea:

The idea behind the proof is illustrated in Fig. 21.11. We define a graph between the plaintexts and ciphertexts, where we put an edge between plaintext x and ciphertext y if there is some key k such that $y = E_k(x)$. The *degree* of this graph is at most the number of potential keys. The fact that the degree is smaller than the number of plaintexts (and hence of ciphertexts) implies that there would be two plaintexts x and x' with different sets of neighbors, and hence the distribution of a ciphertext corresponding to x (with a random key) will not be identical to the distribution of a ciphertext corresponding to x'.

⋆

Proof of Theorem 21.5. Let E, D be a valid encryption scheme with messages of length L and key of length $n < L$. We will show that (E, D) is not perfectly secret by providing two plaintexts $x_0, x_1 \in \{0,1\}^L$ such that the distributions Y_{x_0} and Y_{x_1} are not identical, where Y_x is the distribution obtained by picking $k \sim \{0,1\}^n$ and outputting $E_k(x)$.

We choose $x_0 = 0^L$. Let $S_0 \subseteq \{0,1\}^*$ be the set of all ciphertexts that have non-zero probability of being output in Y_{x_0}. That is, $S_0 = \{y \mid \exists_{k \in \{0,1\}^n} y = E_k(x_0)\}$. Since there are only 2^n keys, we know that $|S_0| \leq 2^n$.

We will show the following claim:

Claim I: There exists some $x_1 \in \{0,1\}^L$ and $k \in \{0,1\}^n$ such that $E_k(x_1) \notin S_0$.

Claim I implies that the string $E_k(x_1)$ has positive probability of being output by Y_{x_1} and zero probability of being output by Y_{x_0} and hence in particular Y_{x_0} and Y_{x_1} are not identical. To prove Claim I, just choose a fixed $k \in \{0,1\}^n$. By the validity condition, the map $x \mapsto E_k(x)$ is a one to one map of $\{0,1\}^L$ to $\{0,1\}^*$ and hence in particular the *image* of this map which is the set $I_k = \{y \mid \exists_{x \in \{0,1\}^L} y = E_k(x)\}$ has size at least (in fact exactly) 2^L. Since $|S_0| \leq 2^n < 2^L$, this means that $|I_k| > |S_0|$ and so in particular there exists some string y in $I_k \setminus S_0$. But by the definition of I_k this means that there is some $x \in \{0,1\}^L$ such that $E_k(x) \notin S_0$ which concludes the proof of Claim I and hence of Theorem 21.5.

■

21.6 COMPUTATIONAL SECRECY

To sum up the previous episodes, we now know that:

- It is possible to obtain a perfectly secret encryption scheme with key length the same as the plaintext.

  and

- It is not possible to obtain such a scheme with key that is even a single bit shorter than the plaintext.

How does this mesh with the fact that, as we've already seen, people routinely use cryptosystems with a 16 byte (i.e., 128 bit) key but many terabytes of plaintext? The proof of Theorem 21.5 does give in fact a way to break all these cryptosystems, but an examination of this proof shows that it only yields an algorithm with time *exponential in the length of the key*. This motivates the following relaxation of perfect secrecy to a condition known as *"computational secrecy"*. Intuitively, an encryption scheme is computationally secret if no polynomial time algorithm can break it. The formal definition is below:

Definition 21.6 — Computational secrecy. Let (E, D) be a valid encryption scheme where for keys of length n, the plaintexts are of length $L(n)$ and the ciphertexts are of length $m(n)$. We say that (E, D) is *computationally secret* if for every polynomial $p : \mathbb{N} \to \mathbb{N}$, and large enough n, if P is an $m(n)$-input and single output NAND-CIRC program of at most $p(n)$ lines, and $x_0, x_1 \in \{0,1\}^{L(n)}$ then

$$\left| \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [P(E_k(x_0))] - \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [P(E_k(x_1))] \right| < \frac{1}{p(n)} \tag{21.3}$$

Definition 21.6 requires a second or third read and some practice to truly understand. One excellent exercise to make sure you follow it is to see that if we allow P to be an *arbitrary* function mapping $\{0,1\}^{m(n)}$ to $\{0,1\}$, and we replace the condition in (21.3) that the left-hand side is smaller than $\frac{1}{p(n)}$ with the condition that it is equal to 0 then we get the perfect secrecy condition of Definition 21.3. Indeed if the distributions $E_k(x_0)$ and $E_k(x_1)$ are identical then applying any function P to them we get the same expectation. On the other hand, if the two distributions above give a different probability for some element $y^* \in \{0,1\}^{m(n)}$, then the function $P(y)$ that outputs 1 iff $y = y^*$ will

have a different expectation under the former distribution than under the latter.

Definition 21.6 raises two natural questions:

- Is it strong enough to ensure that a computationally secret encryption scheme protects the secrecy of messages that are encrypted with it?
- It is weak enough that, unlike perfect secrecy, it is possible to obtain a computationally secret encryption scheme where the key is much smaller than the message?

To the best of our knowledge, the answer to both questions is *Yes*. This is just one example of a much broader phenomenon. We can use computational hardness to achieve many cryptographic goals, including some goals that have been dreamed about for millenia, and other goals that people have not even dared to imagine.

Big Idea 27 *Computational hardness* is *necessary and sufficient* for almost all cryptographic applications.

Regarding the first question, it is not hard to show that if, for example, Alice uses a computationally secret encryption algorithm to encrypt either "attack" or "retreat" (each chosen with probability $1/2$), then as long as she's restricted to polynomial-time algorithms, an adversary Eve will not be able to guess the message with probability better than, say, 0.51, even after observing its encrypted form. (We omit the proof, but it is an excellent exercise for you to work it out on your own.)

To answer the second question we will show that under the same assumption we used for derandomizing **BPP**, we can obtain a computationally secret cryptosystem where the key is almost *exponentially* smaller than the plaintext.

21.6.1 Stream ciphers or the "derandomized one-time pad"

It turns out that if pseudorandom generators exist as in the optimal PRG conjecture, then there exists a computationally secret encryption scheme with keys that are much shorter than the plaintext. The construction below is known as a stream cipher, though perhaps a better name is the "derandomized one-time pad". It is widely used in practice with keys on the order of a few tens or hundreds of bits protecting many terabytes or even petabytes of communication.

We start by recalling the notion of a *pseudorandom generator*, as defined in Definition 20.9. For this chapter, we will fix a special case of the definition:

Figure 21.12: In a *stream cipher* or "derandomized one-time pad" we use a pseudorandom generator $G : \{0,1\}^n \to \{0,1\}^L$ to obtain an encryption scheme with a key length of n and plaintexts of length L. We encrypt the plaintext $x \in \{0,1\}^L$ with key $k \in \{0,1\}^n$ by the ciphertext $x \oplus G(k)$.

Definition 21.7 — Cryptographic pseudorandom generator. Let $L : \mathbb{N} \to \mathbb{N}$ be some function. A *cryptographic pseudorandom generator* with stretch $L(\cdot)$ is a polynomial-time computable function $G : \{0,1\}^* \to \{0,1\}^*$ such that:

- For every $n \in \mathbb{N}$ and $s \in \{0,1\}^n$, $|G(s)| = L(n)$.
- For every polynomial $p : \mathbb{N} \to \mathbb{N}$ and n large enough, if C is a circuit of $L(n)$ inputs, one output, and at most $p(n)$ gates then

$$\left| \Pr_{s \sim \{0,1\}^\ell}[C(G(s)) = 1] - \Pr_{r \sim \{0,1\}^m}[C(r) = 1] \right| < \frac{1}{p(n)} .$$

In this chapter we will call a cryptographic pseudorandom generator simply a *pseudorandom generator* or PRG for short. The optimal PRG conjecture of Section 20.4.2 implies that there is a pseudorandom generator that can "fool" circuits of *exponential size* and where the gap in probabilities is at most one over an exponential quantity. Since exponential grow faster than every polynomial, the optimal PRG conjecture implies the following:

The crypto PRG conjecture: For every $a \in \mathbb{N}$, there is a cryptographic pseudorandom generator with $L(n) = n^a$.

The crypto PRG conjecture is a weaker conjecture than the optimal PRG conjecture, but it too (as we will see) is still stronger than the conjecture that $\mathbf{P} \neq \mathbf{NP}$.

Theorem 21.8 — Derandomized one-time pad. Suppose that the crypto PRG conjecture is true. Then for every constant $a \in \mathbb{N}$ there is a computationally secret encryption scheme (E, D) with plaintext length $L(n)$ at least n^a.

Proof Idea:

The proof is illustrated in Fig. 21.12. We simply take the one-time pad on L bit plaintexts, but replace the key with $G(k)$ where k is a string in $\{0,1\}^n$ and $G : \{0,1\}^n \to \{0,1\}^L$ is a pseudorandom generator. Since the one time pad cannot be broken, an adversary that breaks the derandomized one-time pad can be used to distinguish between the output of the pseudorandom generator and the uniform distribution.

⋆

Proof of Theorem 21.8. Let $G : \{0,1\}^n \to \{0,1\}^L$ for $L = n^a$ be the restriction to input length n of the pseudorandom generator G whose

existence we are guaranteed from the crypto PRG conjecture. We now define our encryption scheme as follows: given key $k \in \{0,1\}^n$ and plaintext $x \in \{0,1\}^L$, the encryption $E_k(x)$ is simply $x \oplus G(k)$. To decrypt a string $y \in \{0,1\}^m$ we output $y \oplus G(k)$. This is a valid encryption since G is computable in polynomial time and $(x \oplus G(k)) \oplus G(k) = x \oplus (G(k) \oplus G(k)) = x$ for every $x \in \{0,1\}^L$.

Computational secrecy follows from the condition of a pseudo-random generator. Suppose, towards a contradiction, that there is a polynomial p, NAND-CIRC program Q of at most $p(L)$ lines and $x, x' \in \{0,1\}^{L(n)}$ such that

$$\left| \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [Q(E_k(x))] - \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [Q(E_k(x'))] \right| > \tfrac{1}{p(L)} \ .$$

(We use here the simple fact that for a $\{0,1\}$-valued random variable X, $\Pr[X = 1] = \mathbb{E}[X]$.)

By the definition of our encryption scheme, this means that

$$\left| \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x)] - \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x')] \right| > \tfrac{1}{p(L)} \ . \tag{21.4}$$

Now since (as we saw in the security analysis of the one-time pad), for every strings $x, x' \in \{0,1\}^L$, the distribution $r \oplus x$ and $r \oplus x'$ are identical, where $r \sim \{0,1\}^L$. Hence

$$\mathop{\mathbb{E}}_{r \sim \{0,1\}^L} [Q(r \oplus x)] = \mathop{\mathbb{E}}_{r \sim \{0,1\}^L} [Q(r \oplus x')] \ . \tag{21.5}$$

By plugging (21.5) into (21.4) we can derive that

$$\left| \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x)] - \mathop{\mathbb{E}}_{r \sim \{0,1\}^L} [Q(r \oplus x)] + \mathop{\mathbb{E}}_{r \sim \{0,1\}^L} [Q(r \oplus x')] - \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x')] \right| > \tfrac{1}{p(L)} \ . \tag{21.6}$$

(Please make sure that you can see why this is true.)

Now we can use the *triangle inequality* that $|A + B| \leq |A| + |B|$ for every two numbers A, B, applying it for $A = \mathbb{E}_{k \sim \{0,1\}^n}[Q(G(k) \oplus x)] - \mathbb{E}_{r \sim \{0,1\}^L}[Q(r \oplus x)]$ and $B = \mathbb{E}_{r \sim \{0,1\}^L}[Q(r \oplus x')] - \mathbb{E}_{k \sim \{0,1\}^n}[Q(G(k) \oplus x')]$ to derive

$$\left| \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x)] - \mathop{\mathbb{E}}_{r \sim \{0,1\}^L} [Q(r \oplus x)] \right| + \left| \mathop{\mathbb{E}}_{r \sim \{0,1\}^L} [Q(r \oplus x')] - \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x')] \right| > \tfrac{1}{p(L)} \ . \tag{21.7}$$

In particular, either the first term or the second term of the left-hand side of (21.7) must be at least $\frac{1}{2p(L)}$. Let us assume the first case holds (the second case is analyzed in exactly the same way). Then we get that

$$\left| \mathop{\mathbb{E}}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x)] - \mathop{\mathbb{E}}_{r \sim \{0,1\}^L} [Q(r \oplus x)] \right| > \tfrac{1}{2p(L)} \ . \tag{21.8}$$

But if we now define the NAND-CIRC program P_x that on input $r \in \{0,1\}^L$ outputs $Q(r \oplus x)$ then (since XOR of L bits can be computed in $O(L)$ lines), we get that P_x has $p(L) + O(L)$ lines and by (21.8) it can distinguish between an input of the form $G(k)$ and an input of the form $r \sim \{0,1\}^k$ with advantage better than $\frac{1}{2p(L)}$. Since a polynomial is dominated by an exponential, if we make L large enough, this will contradict the $(2^{\delta n}, 2^{-\delta n})$ security of the pseudorandom generator G. ■

R

Remark 21.9 — Stream ciphers in practice. The two most widely used forms of (private key) encryption schemes in practice are *stream ciphers* and *block ciphers*. (To make things more confusing, a block cipher is always used in some mode of operation and some of these modes effectively turn a block cipher into a stream cipher.) A block cipher can be thought as a sort of a "random invertible map" from $\{0,1\}^n$ to $\{0,1\}^n$, and can be used to construct a pseudorandom generator and from it a stream cipher, or to encrypt data directly using other modes of operations. There are a great many other security notions and considerations for encryption schemes beyond computational secrecy. Many of those involve handling scenarios such as *chosen plaintext*, *man in the middle*, and *chosen ciphertext* attacks, where the adversary is not just merely a passive eavesdropper but can influence the communication in some way. While this chapter is meant to give you some taste of the ideas behind cryptography, there is much more to know before applying it correctly to obtain secure applications, and a great many people have managed to get it wrong.

21.7 COMPUTATIONAL SECRECY AND NP

We've also mentioned before that an efficient algorithm for **NP** could be used to break all cryptography. We now give an example of how this can be done:

Theorem 21.10 — Breaking encryption using NP algorithm. If $\mathbf{P} = \mathbf{NP}$ then there is no computationally secret encryption scheme with $L(n) > n$.

Furthermore, for every valid encryption scheme (E, D) with $L(n) > n + 100$ there is a polynomial p such that for every large enough n there exist $x_0, x_1 \in \{0,1\}^{L(n)}$ and a $p(n)$-line NAND-

CIRC program *EVE* s.t.

$$\Pr_{i\sim\{0,1\},k\sim\{0,1\}^n}[EVE(E_k(x_i)) = i] \geq 0.99 \ .$$

Note that the "furthermore" part is extremely strong. It means that if the plaintext is even a little bit larger than the key, then we can already break the scheme in a very strong way. That is, there will be a pair of messages x_0, x_1 (think of x_0 as "sell" and x_1 as "buy") and an efficient strategy for Eve such that if Eve gets a ciphertext y then she will be able to tell whether y is an encryption of x_0 or x_1 with probability very close to 1. (We model breaking the scheme as Eve outputting 0 or 1 corresponding to whether the message sent was x_0 or x_1. Note that we could have just as well modified Eve to output x_0 instead of 0 and x_1 instead of 1. The key point is that a priori Eve only had a 50/50 chance of guessing whether Alice sent x_0 or x_1 but after seeing the ciphertext this chance increases to better than 99/1.) The condition $\mathbf{P} = \mathbf{NP}$ can be relaxed to $\mathbf{NP} \subseteq \mathbf{BPP}$ and even the weaker condition $\mathbf{NP} \subseteq \mathbf{P}_{/\mathbf{poly}}$ with essentially the same proof.

Proof Idea:

The proof follows along the lines of Theorem 21.5 but this time paying attention to the computational aspects. If $\mathbf{P} = \mathbf{NP}$ then for every plaintext x and ciphertext y, we can efficiently tell whether there exists $k \in \{0,1\}^n$ such that $E_k(x) = y$. So, to prove this result we need to show that if the plaintexts are long enough, there would exist a pair x_0, x_1 such that the probability that a random encryption of x_1 also is a valid encryption of x_0 will be very small. The details of how to show this are below.

⋆

Proof of Theorem 21.10. We focus on showing only the "furthermore" part since it is the more interesting and the other part follows by essentially the same proof.

Suppose that (E, D) is such an encryption, let n be large enough, and let $x_0 = 0^{L(n)}$. For every $x \in \{0,1\}^{L(n)}$ we define S_x to be the set of all valid encryptions of x. That is $S_x = \{y \mid \exists_{k\in\{0,1\}^n} y = E_k(x)\}$. As in the proof of Theorem 21.5, since there are 2^n keys k, $|S_x| \leq 2^n$ for every $x \in \{0,1\}^{L(n)}$.

We denote by S_0 the set S_{x_0}. We define our algorithm *EVE* to output 0 on input $y \in \{0,1\}^*$ if $y \in S_0$ and to output 1 otherwise. This can be implemented in polynomial time if $\mathbf{P} = \mathbf{NP}$, since the key k can serve the role of an efficiently verifiable solution. (Can you see why?) Clearly $\Pr[EVE(E_k(x_0)) = 0] = 1$ and so in the case that *EVE* gets an encryption of x_0 then she guesses correctly with probability 1. The remainder of the proof is devoted to showing that there ex-

ists $x_1 \in \{0,1\}^{L(n)}$ such that $\Pr[EVE(E_k(x_1)) = 0] \leq 0.01$, which will conclude the proof by showing that *EVE* guesses wrongly with probability at most $\frac{1}{2}0 + \frac{1}{2}0.01 < 0.01$.

Consider now the following probabilistic experiment (which we define solely for the sake of analysis). We consider the sample space of choosing x uniformly in $\{0,1\}^{L(n)}$ and define the random variable $Z_k(x)$ to equal 1 if and only if $E_k(x) \in S_0$. For every k, the map $x \mapsto E_k(x)$ is one-to-one, which means that the probability that $Z_k = 1$ is equal to the probability that $x \in E_k^{-1}(S_0)$ which is $\frac{|S_0|}{2^{L(n)}}$. So by the linearity of expectation $\mathbb{E}[\sum_{k\in\{0,1\}^n} Z_k] \leq \frac{2^n|S_0|}{2^{L(n)}} \leq \frac{2^{2n}}{2^{L(n)}}$.

We will now use the following extremely simple but useful fact known as the *averaging principle* (see also Lemma 18.10): for every random variable Z, if $\mathbb{E}[Z] = \mu$, then with positive probability $Z \leq \mu$. (Indeed, if $Z > \mu$ with probability one, then the expected value of Z will have to be larger than μ, just like you can't have a class in which all students got A or A- and yet the overall average is B+.) In our case it means that with positive probability $\sum_{k\in\{0,1\}^n} Z_k \leq \frac{2^{2n}}{2^{L(n)}}$. In other words, there exists some $x_1 \in \{0,1\}^{L(n)}$ such that $\sum_{k\in\{0,1\}^n} Z_k(x_1) \leq \frac{2^{2n}}{2^{L(n)}}$. Yet this means that if we choose a random $k \sim \{0,1\}^n$, then the probability that $E_k(x_1) \in S_0$ is at most $\frac{1}{2^n} \cdot \frac{2^{2n}}{2^{L(n)}} = 2^{n-L(n)}$. So, in particular if we have an algorithm *EVE* that outputs 0 if $x \in S_0$ and outputs 1 otherwise, then $\Pr[EVE(E_k(x_0)) = 0] = 1$ and $\Pr[EVE(E_k(x_1)) = 0] \leq 2^{n-L(n)}$ which will be smaller than $2^{-10} < 0.01$ if $L(n) \geq n + 10$.

■

In retrospect Theorem 21.10 is perhaps not surprising. After all, as we've mentioned before it is known that the Optimal PRG conjecture (which is the basis for the derandomized one-time pad encryption) is *false* if $\mathbf{P} = \mathbf{NP}$ (and in fact even if $\mathbf{NP} \subseteq \mathbf{BPP}$ or even $\mathbf{NP} \subseteq \mathbf{P}_{/\mathbf{poly}}$).

21.8 PUBLIC KEY CRYPTOGRAPHY

People have been dreaming about heavier-than-air flight since at least the days of Leonardo Da Vinci (not to mention Icarus from the greek mythology). Jules Verne wrote with rather insightful details about going to the moon in 1865. But, as far as I know, in all the thousands of years people have been using secret writing, until about 50 years ago no one has considered the possibility of communicating securely without first exchanging a shared secret key.

Yet in the late 1960's and early 1970's, several people started to question this "common wisdom". Perhaps the most surprising of these visionaries was an undergraduate student at Berkeley named Ralph Merkle. In the fall of 1974 Merkle wrote in a project proposal for his computer security course that while "it might seem intuitively

obvious that if two people have never had the opportunity to prearrange an encryption method, then they will be unable to communicate securely over an insecure channel... I believe it is false". The project proposal was rejected by his professor as "not good enough". Merkle later submitted a paper to the communication of the ACM where he apologized for the lack of references since he was unable to find any mention of the problem in the scientific literature, and the only source where he saw the problem even *raised* was in a science fiction story. The paper was rejected with the comment that "Experience shows that it is extremely dangerous to transmit key information in the clear." Merkle showed that one can design a protocol where Alice and Bob can use T invocations of a hash function to exchange a key, but an adversary (in the random oracle model, though he of course didn't use this name) would need roughly T^2 invocations to break it. He conjectured that it may be possible to obtain such protocols where breaking is *exponentially harder* than using them, but could not think of any concrete way to doing so.

We only found out much later that in the late 1960's, a few years before Merkle, James Ellis of the British Intelligence agency GCHQ was having similar thoughts. His curiosity was spurred by an old World-War II manuscript from Bell Labs that suggested the following way that two people could communicate securely over a phone line. Alice would inject noise to the line, Bob would relay his messages, and then Alice would subtract the noise to get the signal. The idea is that an adversary over the line sees only the sum of Alice's and Bob's signals, and doesn't know what came from what. This got James Ellis thinking whether it would be possible to achieve something like that digitally. As Ellis later recollected, in 1970 he realized that in principle this should be possible, since he could think of an hypothetical black box B that on input a "handle" α and plaintext x would give a "ciphertext" y and that there would be a secret key β corresponding to α, such that feeding β and y to the box would recover x. However, Ellis had no idea how to actually instantiate this box. He and others kept giving this question as a puzzle to bright new recruits until one of them, Clifford Cocks, came up in 1973 with a candidate solution loosely based on the factoring problem; in 1974 another GCHQ recruit, Malcolm Williamson, came up with a solution using modular exponentiation.

But among all those thinking of public key cryptography, probably the people who saw the furthest were two researchers at Stanford, Whit Diffie and Martin Hellman. They realized that with the advent of electronic communication, cryptography would find new applications beyond the military domain of spies and submarines, and they understood that in this new world of many users and point to point

communication, cryptography will need to scale up. Diffie and Hellman envisioned an object which we now call "trapdoor permutation" though they called "one way trapdoor function" or sometimes simply "public key encryption". Though they didn't have full formal definitions, their idea was that this is an injective function that is easy (e.g., polynomial-time) to *compute* but hard (e.g., exponential-time) to *invert*. However, there is a certain *trapdoor*, knowledge of which would allow polynomial time inversion. Diffie and Hellman argued that using such a trapdoor function, it would be possible for Alice and Bob to communicate securely *without ever having exchanged a secret key*. But they didn't stop there. They realized that protecting the *integrity* of communication is no less important than protecting its *secrecy*. Thus they imagined that Alice could "run encryption in reverse" in order to certify or *sign* messages.

At the point, Diffie and Hellman were in a position not unlike physicists who predicted that a certain particle should exist but without any experimental verification. Luckily they met Ralph Merkle, and his ideas about a probabilistic *key exchange protocol*, together with a suggestion from their Stanford colleague John Gill, inspired them to come up with what today is known as the Diffie Hellman Key Exchange (which unbeknownst to them was found two years earlier at GCHQ by Malcolm Williamson). They published their paper "New Directions in Cryptography" in 1976, and it is considered to have brought about the birth of modern cryptography.

The Diffie-Hellman Key Exchange is still widely used today for secure communication. However, it still felt short of providing Diffie and Hellman's elusive trapdoor function. This was done the next year by Rivest, Shamir and Adleman who came up with the RSA trapdoor function, which through the framework of Diffie and Hellman yielded not just encryption but also signatures. (A close variant of the RSA function was discovered earlier by Clifford Cocks at GCHQ, though as far as I can tell Cocks, Ellis and Williamson did not realize the application to digital signatures.) From this point on began a flurry of advances in cryptography which hasn't died down till this day.

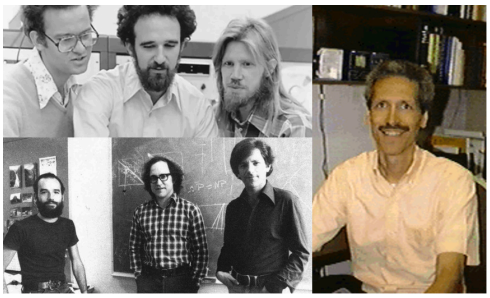


Figure 21.13: Top left: Ralph Merkle, Martin Hellman and Whit Diffie, who together came up in 1976 with the concept of *public key encryption* and a *key exchange protocol*. Bottom left: Adi Shamir, Ron Rivest, and Leonard Adleman who, following Diffie and Hellman's paper, discovered the RSA function that can be used for public key encryption and digital signatures. Interestingly, one can see the equation **P** = **NP** on the blackboard behind them. Right: John Gill, who was the first person to suggest to Diffie and Hellman that they use modular exponentiation as an easy-to-compute but hard-to-invert function.

21.8.1 Defining public key encryption

A *public key encryption* consists of a triple of algorithms:

- The *key generation algorithm*, which we denote by $KeyGen$ or KG for short, is a randomized algorithm that outputs a pair of strings (e, d) where e is known as the *public* (or *encryption*) key, and d is known as the *private* (or *decryption*) key. The key generation algorithm gets as input 1^n (i.e., a string of ones of length n). We refer to n as the *security parameter* of the scheme. The bigger we make n, the more secure the encryption will be, but also the less efficient it will be.

- The *encryption algorithm*, which we denote by E, takes the encryption key e and a plaintext x, and outputs the ciphertext $y = E_e(x)$.
- The *decryption algorithm*, which we denote by D, takes the decryption key d and a ciphertext y, and outputs the plaintext $x = D_d(y)$.

We now make this a formal definition:

Definition 21.11 — Public Key Encryption. A *computationally secret public key encryption* with plaintext length $L : \mathbb{N} \to \mathbb{N}$ is a triple of randomized polynomial-time algorithms (KG, E, D) that satisfy the following conditions:

- For every n, if (e, d) is output by $KG(1^n)$ with positive probability, and $x \in \{0,1\}^{L(n)}$, then $D_d(E_e(x)) = x$ with probability one.
- For every polynomial p, and sufficiently large n, if P is a NAND-CIRC program of at most $p(n)$ lines then for every $x, x' \in \{0,1\}^{L(n)}$, $|\mathbb{E}[P(e, E_e(x))] - \mathbb{E}[P(e, E_e(x'))]| < 1/p(n)$, where this probability is taken over the coins of KG and E.

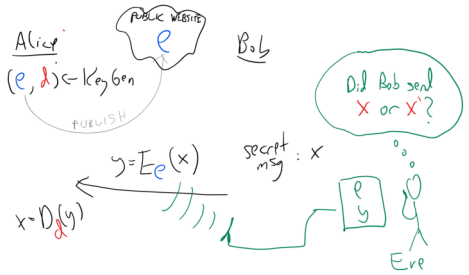


Figure 21.14: In a *public key encryption*, Alice generates a private/public keypair (e, d), publishes e and keeps d secret. To encrypt a message for Alice, one only needs to know e. To decrypt it we need to know d.

Definition 21.11 allows E and D to be *randomized* algorithms. In fact, it turns out that it is *necessary* for E to be randomized to obtain computational secrecy. It also turns out that, unlike the private key case, we can transform a public-key encryption that works for messages that are *only one bit long* into a public-key encryption scheme that can encrypt arbitrarily long messages, and in particular messages that are *longer than the key*. In particular this means that we cannot obtain a perfectly secret public-key encryption scheme even for one-bit long messages (since it would imply a perfectly secret public-key, and hence in particular private-key, encryption with messages longer than the key).

We will not give full constructions for public key encryption schemes in this chapter, but will mention some of the ideas that underlie the most widely used schemes today. These generally belong to one of two families:

- *Group theoretic constructions* based on problems such as *integer factoring* and the *discrete logarithm* over finite fields or elliptic curves.
- *Lattice/coding based constructions* based on problems such as the *closest vector in a lattice* or *bounded distance decoding*.

Group-theory based encryptions such as the RSA cryptosystem, the Diffie-Hellman protocol, and Elliptic-Curve Cryptography, are currently more widely implemented. But the lattice/coding schemes are

recently on the rise, particularly because the known group theoretic encryption schemes can be broken by *quantum computers*, which we discuss in Chapter 23.

21.8.2 Diffie-Hellman key exchange

As just one example of how public key encryption schemes are constructed, let us now describe the Diffie-Hellman key exchange. We describe the Diffie-Hellman protocol in a somewhat of an informal level, without presenting a full security analysis.

The computational problem underlying the Diffie Hellman protocol is the *discrete logarithm problem*. Let's suppose that g is some integer. We can compute the map $x \mapsto g^x$ and also its *inverse* $y \mapsto \log_g y$. (For example, we can compute a logarithm is by *binary search*: start with some interval $[x_{min}, x_{max}]$ that is guaranteed to contain $\log_g y$. We can then test whether the interval's midpoint x_{mid} satisfies $g^{x_{mid}} > y$, and based on that halve the size of the interval.)

However, suppose now that we use *modular arithmetic* and work modulo some prime number p. If p has n binary digits and g is in $[p]$ then we can compute the map $x \mapsto g^x \mod p$ in time polynomial in n. (This is not trivial, and is a great exercise for you to work this out; as a hint, start by showing that one can compute the map $k \mapsto g^{2^k} \mod p$ using k modular multiplications modulo p, if you're stumped, you can look up this Wikipedia entry.) On the other hand, because of the "wraparound" property of modular arithmetic, we cannot run binary search to find the inverse of this map (known as the *discrete logarithm*). In fact, there is no known polynomial-time algorithm for computing this discrete logarithm map $(g, x, p) \mapsto \log_g x \mod p$, where we define $\log_g x \mod p$ as the number $a \in [p]$ such that $g^a = x \mod p$.

The Diffie-Hellman protocol for Bob to send a message to Alice is as follows:

- **Alice:** Chooses p to be a random n bit long prime (which can be done by choosing random numbers and running a primality testing algorithm on them), and g and a at random in $[p]$. She sends to Bob the triple $(p, g, g^a \mod p)$.

- **Bob:** Given the triple (p, g, h), Bob sends a message $x \in \{0,1\}^L$ to Alice by choosing b at random in $[p]$, and sending to Alice the pair $(g^b \mod p, rep(h^b \mod p) \oplus x)$ where $rep : [p] \to \{0,1\}^*$ is some "representation function" that maps $[p]$ to $\{0,1\}^L$. (The function rep does not need to be one-to-one and you can think of $rep(z)$ as simply outputting L of the bits of z in the natural binary representation, it does need to satisfy certain technical conditions which we omit in this description.)

- **Alice:** Given g', z, Alice recovers x by outputting $rep(g'^a \mod p) \oplus z$.

The correctness of the protocol follows from the simple fact that $(g^a)^b = (g^b)^a$ for every g, a, b and this still holds if we work modulo p. Its security relies on the computational assumption that computing this map is hard, even in a certain "average case" sense (this computational assumption is known as the Decisional Diffie Hellman assumption). The Diffie-Hellman key exchange protocol can be thought of as a public key encryption where Alice's first message is the public key, and Bob's message is the encryption.

One can think of the Diffie-Hellman protocol as being based on a "trapdoor pseudorandom generator" where the triple g^a, g^b, g^{ab} looks "random" to someone that doesn't know a, but someone that does know a can see that raising the second element to the a-th power yields the third element. The Diffie-Hellman protocol can be described abstractly in the context of any finite Abelian group for which we can efficiently compute the group operation. It has been implemented on other groups than numbers modulo p, and in particular Elliptic Curve Cryptography (ECC) is obtained by basing the Diffie Hellman on elliptic curve groups which gives some practical advantages. Another common group theoretic basis for key-exchange/public key encryption protocol is the RSA function. A big disadvantage of Diffie-Hellman (both the modular arithmetic and elliptic curve variants) and RSA is that both schemes can be broken in polynomial time by a *quantum computer*. We will discuss quantum computing later in this course.

21.9 OTHER SECURITY NOTIONS

There is a great deal to cryptography beyond just encryption schemes, and beyond the notion of a passive adversary. A central objective is *integrity* or *authentication*: protecting communications from being modified by an adversary. Integrity is often more fundamental than secrecy: whether it is a software update or viewing the news, you might often not care about the communication being secret as much as that it indeed came from its claimed source. *Digital signature schemes* are the analog of public key encryption for authentication, and are widely used (in particular as the basis for public key certificates) to provide a foundation of trust in the digital world.

Similarly, even for encryption, we often need to ensure security against *active attacks*, and so notions such as non-malleability and adaptive chosen ciphertext security have been proposed. An encryption scheme is only as secure as the secret key, and mechanisms to make sure the key is generated properly, and is protected against re-

fresh or even compromise (i.e., forward secrecy) have been studied as well. Hopefully this chapter provides you with some appreciation for cryptography as an intellectual field, but does not imbue you with a false self confidence in implementing it.

Cryptographic hash functions are another widely used tool with a variety of uses, including extracting randomness from high entropy sources, achieving hard-to-forge short "digests" of files, protecting passwords, and much more.

21.10 MAGIC

Beyond encryption and signature schemes, cryptographers have managed to obtain objects that truly seem paradoxical and "magical". We briefly discuss some of these objects. We do not give any details, but hopefully this will spark your curiosity to find out more.

21.10.1 Zero knowledge proofs

On October 31, 1903, the mathematician Frank Nelson Cole gave an hourlong lecture to a meeting of the American Mathematical Society where he did not speak a single word. Rather, he calculated on the board the value $2^{67} - 1$ which is equal to $147,573,952,589,676,412,927$, and then showed that this number is equal to $193,707,721 \times 761,838,257,287$. Cole's proof showed that $2^{67} - 1$ is not a prime, but it also revealed additional information, namely its actual factors. This is often the case with proofs: they teach us more than just the validity of the statements.

In *Zero Knowledge Proofs* we try to achieve the opposite effect. We want a proof for a statement X where we can *rigorously show* that the proofs reveals *absolutely no additional information about* X beyond the fact that it is true. This turns out to be an extremely useful object for a variety of tasks including authentication, secure protocols, voting, anonymity in cryptocurrencies, and more. Constructing these objects relies on the theory of **NP** completeness. Thus this theory that originally was designed to give a *negative result* (show that some problems are hard) ended up yielding *positive applications*, enabling us to achieve tasks that were not possible otherwise.

21.10.2 Fully homomorphic encryption

Suppose that we are given a bit-by-bit encryption of a string $E_k(x_0), \ldots, E_k(x_{n-1})$. By design, these ciphertexts are supposed to be "completely unscrutable" and we should not be able to extract any information about x_i's from it. However, already in 1978, Rivest, Adleman and Dertouzos observed that this does not imply that we could not *manipulate* these encryptions. For example, it turns out the security of an encryption scheme does not immediately rule out the

ability to take a pair of encryptions $E_k(a)$ and $E_k(b)$ and compute from them $E_k(a NAND b)$ *without knowing the secret key* k. But do there exist encryption schemes that allow such manipulations? And if so, is this a bug or a feature?

Rivest et al already showed that such encryption schemes could be *immensely* useful, and their utility has only grown in the age of cloud computing. After all, if we can compute NAND then we can use this to run any algorithm P on the encrypted data, and map $E_k(x_0), \ldots, E_k(x_{n-1})$ to $E_k(P(x_0, \ldots, x_{n-1}))$. For example, a client could store their secret data x in encrypted form on the cloud, and have the cloud provider perform all sorts of computation on these data without ever revealing to the provider the private key, and so without the provider *ever learning any information* about the secret data.

The question of *existence* of such a scheme took much longer time to resolve. Only in 2009 Craig Gentry gave the first construction of an encryption scheme that allows to compute a universal basis of gates on the data (known as a *Fully Homomorphic Encryption scheme* in crypto parlance). Gentry's scheme left much to be desired in terms of efficiency, and improving upon it has been the focus of an intensive research program that has already seen significant improvements.

21.10.3 Multiparty secure computation

Cryptography is about enabling mutually distrusting parties to achieve a common goal. Perhaps the most general primitive achieving this objective is secure multiparty computation. The idea in secure multiparty computation is that n parties interact together to compute some function $F(x_0, \ldots, x_{n-1})$ where x_i is the private input of the i-th party. The crucial point is that there is *no commonly trusted party or authority* and that nothing is revealed about the secret data beyond the function's output. One example is an *electronic voting protocol* where only the total vote count is revealed, with the privacy of the individual voters protected, but without having to trust any authority to either count the votes correctly or to keep information confidential. Another example is implementing a second price (aka Vickrey) auction where $n-1$ parties submit bids to an item owned by the n-th party, and the item goes to the highest bidder but at the price of the *second highest bid*. Using secure multiparty computation we can implement second price auction in a way that will ensure the secrecy of the numerical values of all bids (including even the top one) except the second highest one, and the secrecy of the identity of all bidders (including even the second highest bidder) except the top one. We emphasize that such a protocol requires no trust even in the auctioneer itself, who will also not learn any additional information. Secure multiparty computation

can be used even for computing *randomized* processes, with one example being playing Poker over the net without having to trust any server for correct shuffling of cards or not revealing the information.

Chapter Recap

- We can formally define the notion of security of an encryption scheme.
- *Perfect secrecy* ensures that an adversary does not learn *anything* about the plaintext from the ciphertext, regardless of their computational powers.
- The one-time pad is a perfectly secret encryption with the length of the key equaling the length of the message. No perfectly secret encryption can have key shorter than the message.
- *Computational secrecy* can be as good as perfect secrecy since it ensures that the advantage that computationally bounded adversaries gain from observing the ciphertext is exponentially small. If the optimal PRG conjecture is true then there exists a computationally secret encryption scheme with messages that can be (almost) *exponentially bigger* than the key.
- There are many cryptographic tools that go well beyond private key encryption. These include *public key encryption*, *digital signatures* and *hash functions*, as well as more "magical" tools such as *multiparty secure computation*, *fully homomorphic encryption*, *zero knowledge proofs*, and many others.

21.11 EXERCISES

21.12 BIBLIOGRAPHICAL NOTES

Much of this text is taken from my lecture notes on cryptography.

Shannon's manuscript was written in 1945 but was classified, and a partial version was only published in 1949. Still it has revolutionized cryptography, and is the forerunner to much of what followed.

The Venona project's history is described in this document. Aside from Grabeel and Zubko, credit to the discovery that the Soviets were reusing keys is shared by Lt. Richard Hallock, Carrie Berry, Frank Lewis, and Lt. Karl Elmquist, and there are others that have made important contribution to this project. See pages 27 and 28 in the document.

In a 1955 letter to the NSA that only recently came forward, John Nash proposed an "unbreakable" encryption scheme. He wrote *"I hope my handwriting, etc. do not give the impression I am just a crank or circle-squarer.... The significance of this conjecture* [*that certain encryption*

schemes are exponentially secure against key recovery attacks] .. *is that it is quite feasible to design ciphers that are effectively unbreakable.*". John Nash made seminal contributions in mathematics and game theory, and was awarded both the Abel Prize in mathematics and the Nobel Memorial Prize in Economic Sciences. However, he has struggled with mental illness throughout his life. His biography, A Beautiful Mind was made into a popular movie. It is natural to compare Nash's 1955 letter to the NSA to Gödel's letter to von Neumann we mentioned before. From the theoretical computer science point of view, the crucial difference is that while Nash informally talks about exponential vs polynomial computation time, he does not mention the word "Turing machine" or other models of computation, and it is not clear if he is aware or not that his conjecture can be made mathematically precise (assuming a formalization of "sufficiently complex types of enciphering").

The definition of computational secrecy we use is the notion of *computational indistinguishability* (known to be equivalent to *semantic security*) that was given by Goldwasser and Micali in 1982.

Although they used a different terminology, Diffie and Hellman already made clear in their paper that their protocol can be used as a public key encryption, with the first message being put in a "public file". In 1985, ElGamal showed how to obtain a *signature scheme* based on the Diffie Hellman ideas, and since he described the Diffie-Hellman encryption scheme in the same paper, the public key encryption scheme originally proposed by Diffie and Hellman is sometimes also known as ElGamal encryption.

My survey contains a discussion on the different types of public key assumptions. While the standard elliptic curve cryptographic schemes are as susceptible to quantum computers as Diffie-Hellman and RSA, their main advantage is that the best known classical algorithms for computing discrete logarithms over elliptic curve groups take time $2^{\epsilon n}$ for some $\epsilon > 0$ where n is the number of bits to describe a group element. In contrast, for the multiplicative group modulo a prime p the best algorithm take time $2^{O(n^{1/3}polylog(n))}$ which means that (assuming the known algorithms are optimal) we need to set the prime to be bigger (and so have larger key sizes with corresponding overhead in communication and computation) to get the same level of security.

Zero-knowledge proofs were constructed by Goldwasser, Micali, and Rackoff in 1982, and their wide applicability was shown (using the theory of **NP** completeness) by Goldreich, Micali, and Wigderson in 1986.

Two party and multiparty secure computation protocols were constructed (respectively) by Yao in 1982 and Goldreich, Micali, and Wigderson in 1987. The latter work gave a general transformation

from security against passive adversaries to security against active adversaries using zero knowledge proofs.

22
Proofs and algorithms

> *"Let's not try to define knowledge, but try to define zero-knowledge."*, Shafi Goldwasser.

Proofs have captured human imagination for thousands of years, ever since the publication of Euclid's *Elements*, a book second only to the bible in the number of editions printed.

Plan:

- Proofs and algorithms
- Interactive proofs
- Zero knowledge proofs
- Propositions as types, Coq and other proof assistants.

22.1 EXERCISES

22.2 BIBLIOGRAPHICAL NOTES

Learning Objectives:

- **See main aspects in which quantum mechanics differs from local deterministic theories.**
- **Model of quantum circuits, or equivalently QNAND-CIRC programs**
- **The complexity class BQP and what we know about its relation to other classes**
- **Ideas behind Shor's Algorithm and the Quantum Fourier Transform**

23
Quantum computing

> *"We always have had (secret, secret, close the doors!) ... a great deal of difficulty in understanding the world view that quantum mechanics represents ... It has not yet become obvious to me that there's no real problem. ... Can I learn anything from asking this question about computers–about this may or may not be mystery as to what the world view of quantum mechanics is?"* , Richard Feynman, 1981

> *"The only difference between a probabilistic classical world and the equations of the quantum world is that somehow or other it appears as if the probabilities would have to go negative"*, Richard Feynman, 1981

There were two schools of natural philosophy in ancient Greece. *Aristotle* believed that objects have an *essence* that explains their behavior, and a theory of the natural world has to refer to the *reasons* (or "final cause" to use Aristotle's language) as to why they exhibit certain phenomena. *Democritus* believed in a purely mechanistic explanation of the world. In his view, the universe was ultimately composed of elementary particles (or *Atoms*) and our observed phenomena arise from the interactions between these particles according to some local rules. Modern science (arguably starting with Newton) has embraced Democritus' point of view, of a mechanistic or "clockwork" universe of particles and forces acting upon them.

While the classification of particles and forces evolved with time, to a large extent the "big picture" has not changed from Newton till Einstein. In particular it was held as an axiom that if we knew fully the current *state* of the universe (i.e., the particles and their properties such as location and velocity) then we could predict its future state at any point in time. In computational language, in all these theories the state of a system with n particles could be stored in an array of $O(n)$ numbers, and predicting the evolution of the system can be done by running some efficient (e.g., $poly(n)$ time) deterministic computation on this array.

23.1 THE DOUBLE SLIT EXPERIMENT

Alas, in the beginning of the 20th century, several experimental results were calling into question this "clockwork" or "billiard ball" theory of the world. One such experiment is the famous double slit experiment. Here is one way to describe it. Suppose that we buy one of those baseball pitching machines, and aim it at a soft plastic wall, but put a *metal barrier with a single slit* between the machine and the plastic wall (see Fig. 23.1). If we shoot baseballs at the plastic wall, then some of the baseballs would bounce off the metal barrier, while some would make it through the slit and dent the wall. If we now carve out an additional slit in the metal barrier then more balls would get through, and so the plastic wall would be *even more dented*.

So far this is pure common sense, and it is indeed (to my knowledge) an accurate description of what happens when we shoot baseballs at a plastic wall. However, this is not the same when we shoot *photons*. Amazingly, if we shoot with a "photon gun" (i.e., a laser) at a wall equipped with photon detectors through some barrier, then (as shown in Fig. 23.2) in some positions of the wall we will see *fewer* hits when the two slits are open than when only one of them is.[1] In particular there are positions in the wall that are hit when the first slit is open, hit when the second slit is open, but are *not hit at all when both slits are open!*

It seems as if each photon coming out of the gun is aware of the global setup of the experiment, and behaves differently if two slits are open than if only one is. If we try to "catch the photon in the act" and place a detector right next to each slit so we can see exactly the path each photon takes then something even more bizarre happens. The mere fact that we *measure* the path changes the photon's behavior, and now this "destructive interference" pattern is gone and the number of times a position is hit when two slits are open is the sum of the number of times it is hit when each slit is open.

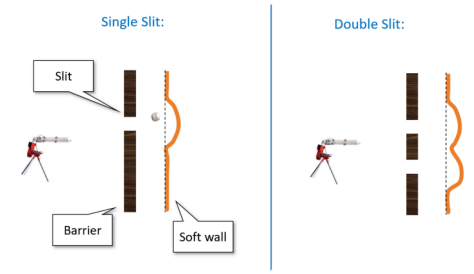


Figure 23.1: In the "double baseball experiment" we shoot baseballs from a gun at a soft wall through a hard barrier that has one or two slits open in it. There is only "constructive interference" in the sense that the dent in each position in the wall when both slits are open is the sum of the dents when each slit is open on its own.

[1] A nice illustrated description of the double slit experiment appears in this video.

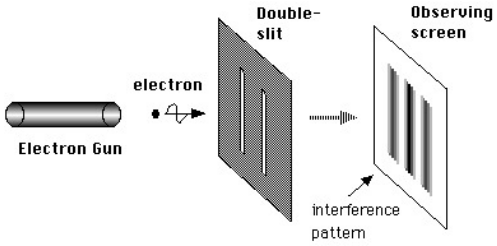


Figure 23.2: The setup of the double slit experiment in the case of photon or electron guns. We see also *destructive* interference in the sense that there are some positions on the wall that get *fewer* hits when both slits are open than they get when only one of the slits is open. Image credit: Wikipedia.

P You should read the paragraphs above more than once and make sure you appreciate how truly mind boggling these results are.

23.2 QUANTUM AMPLITUDES

The double slit and other experiments ultimately forced scientists to accept a very counterintuitive picture of the world. It is not merely about nature being randomized, but rather it is about the probabilities in some sense "going negative" and cancelling each other!

To see what we mean by this, let us go back to the baseball experiment. Suppose that the probability a ball passes through the left slit is p_L and the probability that it passes through the right slit is p_R. Then, if we shoot N balls out of each gun, we expect the wall will be hit $(p_L + p_R)N$ times. In contrast, in the quantum world of photons instead of baseballs, it can sometimes be the case that in both the first and second case the wall is hit with positive probabilities p_L and p_R respectively but somehow when both slits are open the wall (or a particular position in it) is not hit at all. It's almost as if the probabilities can "cancel each other out".

To understand the way we model this in quantum mechanics, it is helpful to think of a "lazy evaluation" approach to probability. We can think of a probabilistic experiment such as shooting a baseball through two slits in two different ways:

- When a ball is shot, "nature" tosses a coin and decides if it will go through the left slit (which happens with probability p_L), right slit (which happens with probability p_R), or bounce back. If it passes through one of the slits then it will hit the wall. Later we can look at the wall and find out whether or not this event happened, but the fact that the event happened or not is determined independently of whether or not we look at the wall.

- The other viewpoint is that when a ball is shot, "nature" computes the probabilities p_L and p_R as before, but does *not* yet "toss the coin" and determines what happened. Only when we actually look at the wall, nature tosses a coin and with probability $p_L + p_R$ ensures we see a dent. That is, nature uses "lazy evaluation", and only determines the result of a probabilistic experiment when we decide to *measure* it.

While the first scenario seems much more natural, the end result in both is the same (the wall is hit with probability $p_L + p_R$) and so the question of whether we should model nature as following the first scenario or second one seems like asking about the proverbial tree that falls in the forest with no one hearing it.

However, when we want to describe the double slit experiment with photons rather than baseballs, it is the second scenario that lends itself better to a quantum generalization. Quantum mechanics associates a number α known as an *amplitude* with each probabilistic experiment. This number α can be *negative*, and in fact even *complex*. We never observe the amplitudes directly, since whenever we *measure* an event with amplitude α, nature tosses a coin and determines that the event happens with probability $|\alpha|^2$. However, the sign (or in the complex case, phase) of the amplitudes can affect whether two different events have *constructive* or *destructive* interference.

Specifically, consider an event that can either occur or not (e.g. "detector number 17 was hit by a photon"). In classical probability, we model this by a probability distribution over the two outcomes: a pair of non-negative numbers p and q such that $p + q = 1$, where p corresponds to the probability that the event occurs and q corresponds to the probability that the event does not occur. In quantum mechanics, we model this also by pair of numbers, which we call *amplitudes*. This is a pair of (potentially negative or even complex) numbers α and β such that $|\alpha|^2 + |\beta|^2 = 1$. The probability that the event occurs is $|\alpha|^2$ and the probability that it does not occur is $|\beta|^2$. In isolation, these negative or complex numbers don't matter much, since we square them anyway to obtain probabilities. But the interaction of positive and negative amplitudes can result in surprising *cancellations* where somehow combining two scenarios where an event happens with positive probability results in a scenario where it never does.

If you don't find the above description confusing and unintuitive, you probably didn't get it. Please make sure to re-read the above paragraphs until you are thoroughly confused.

Quantum mechanics is a mathematical theory that allows us to calculate and predict the results of the double-slit and many other experiments. If you think of quantum mechanics as an explanation as to what "really" goes on in the world, it can be rather confusing. However, if you simply "shut up and calculate" then it works amazingly well at predicting experimental results. In particular, in the double slit experiment, for any position in the wall, we can compute numbers α and β such that photons from the first and second slit hit that position with probabilities $|\alpha|^2$ and $|\beta|^2$ respectively. When we open both slits, the probability that the position will be hit is proportional to $|\alpha + \beta|^2$, and so in particular, if $\alpha = -\beta$ then it will be the case that, despite being hit when *either* slit one or slit two are open, the position is *not hit at all* when they both are. If you are confused by quantum mechanics, you are not alone: for decades people have been trying to come up with explanations for "the underlying reality" behind quantum mechanics, including Bohmian Mechanics, Many Worlds and others. However, none of these interpretations have gained universal acceptance and all of those (by design) yield the same experimental predictions. Thus at this point many scientists prefer to just ignore the question of what is the "true reality" and go back to simply "shutting up and calculating".

R

Remark 23.1 — Complex vs real, other simplifications. If (like the author) you are a bit intimidated by complex numbers, don't worry: you can think of all amplitudes as *real* (though potentially *negative*) numbers without loss of understanding. All the "magic" of quantum computing already arises in this case, and so we will often restrict attention to real amplitudes in this chapter.

We will also only discuss so-called *pure* quantum states, and not the more general notion of *mixed* states. Pure states turn out to be sufficient for understanding the algorithmic aspects of quantum computing.

More generally, this chapter is not meant to be a complete description of quantum mechanics, quantum information theory, or quantum computing, but rather illustrate the main points where these differ from classical computing.

23.3 BELL'S INEQUALITY

There is something weird about quantum mechanics. In 1935 Einstein, Podolsky and Rosen (EPR) tried to pinpoint this issue by highlighting a previously unrealized corollary of this theory. They showed that the idea that nature does not determine the results of an experiment until it is measured results in so called "spooky action at a distance". Namely, making a measurement of one object may instantaneously effect the state (i.e., the vector of amplitudes) of another object in the other end of the universe.

Since the vector of amplitudes is just a mathematical abstraction, the EPR paper was considered to be merely a thought experiment for philosophers to be concerned about, without bearing on experiments. This changed when in 1965 John Bell showed an actual experiment to test the predictions of EPR and hence pit intuitive common sense against the quantum mechanics. Quantum mechanics won: it turns out that it *is* in fact possible to use measurements to create correlations between the states of objects far removed from one another that cannot be explained by any prior theory. Nonetheless, since the results of these experiments are so obviously wrong to anyone that has ever sat in an armchair, that there are still a number of Bell denialists arguing that this can't be true and quantum mechanics is wrong.

So, what is this Bell's Inequality? Suppose that Alice and Bob try to convince you they have telepathic ability, and they aim to prove it via the following experiment. Alice and Bob will be in separate closed rooms.[2] You will interrogate Alice and your associate will interrogate Bob. You choose a random bit $x \in \{0,1\}$ and your associate chooses

[2] If you are extremely paranoid about Alice and Bob communicating with one another, you can coordinate with your assistant to perform the experiment exactly at the same time, and make sure that the rooms are sufficiently far apart (e.g., are on two different continents, or maybe even one is on the moon and another is on earth) so that Alice and Bob couldn't communicate to each other in time even if they do so at the speed of light.

a random $y \in \{0,1\}$. We let a be Alice's response and b be Bob's response. We say that Alice and Bob win this experiment if $a \oplus b = x \wedge y$. In other words, Alice and Bob need to output two bits that *disagree* if $x = y = 1$ and *agree* otherwise.[3]

[3] This form of Bell's game was shown by Clauser, Horne, Shimony, and Holt.

Now if Alice and Bob are not telepathic, then they need to agree in advance on some strategy. It's not hard for Alice and Bob to succeed with probability $3/4$: just always output the same bit. Moreover, by doing some case analysis, we can show that no matter what strategy they use, Alice and Bob cannot succeed with higher probability than that:[4]

[4] Theorem 23.2 below assumes that Alice and Bob use *deterministic* strategies f and g respectively. More generally, Alice and Bob could use a *randomized* strategy, or equivalently, each could choose f and g from some *distributions* $\mathcal{F}$ and $\mathcal{G}$ respectively. However the *averaging principle* (Lemma 18.10) implies that if all possible deterministic strategies succeed with probability at most $3/4$, then the same is true for all randomized strategies.

Theorem 23.2 — Bell's Inequality. For every two functions $f, g : \{0,1\} \to \{0,1\}$, $\Pr_{x,y\in\{0,1\}}[f(x) \oplus g(y) = x \wedge y] \leq 3/4$.

Proof. Since the probability is taken over all four choices of $x, y \in \{0,1\}$, the only way the theorem can be violated if if there exist two functions f, g that satisfy

$$f(x) \oplus g(y) = x \wedge y$$

for all the four choices of $x, y \in \{0,1\}^2$. Let's plug in all these four choices and see what we get (below we use the equalities $z \oplus 0 = z$, $z \wedge 0 = 0$ and $z \wedge 1 = z$):

$$\begin{aligned} f(0) \oplus g(0) &= 0 \quad \text{(plugging in } x = 0, y = 0) \\ f(0) \oplus g(1) &= 0 \quad \text{(plugging in } x = 0, y = 1) \\ f(1) \oplus g(0) &= 0 \quad \text{(plugging in } x = 1, y = 0) \\ f(1) \oplus g(1) &= 1 \quad \text{(plugging in } x = 1, y = 1) \end{aligned}$$

If we XOR together the first and second equalities we get $g(0) \oplus g(1) = 0$ while if we XOR together the third and fourth equalities we get $g(0) \oplus g(1) = 1$, thus obtaining a contradiction. ■

An amazing experimentally verified fact is that quantum mechanics allows for "telepathy".[5] Specifically, it has been shown that using the weirdness of quantum mechanics, there is in fact a strategy for Alice and Bob to succeed in this game with probability larger than $3/4$ (in fact, they can succeed with probability about 0.85, see Lemma 23.5).

[5] More accurately, one either has to give up on a "billiard ball type" theory of the universe or believe in telepathy (believe it or not, some scientists went for the latter option).

23.4 QUANTUM WEIRDNESS

Some of the counterintuitive properties that arise from quantum mechanics include:

- **Interference** - As we've seen, quantum amplitudes can "cancel each other out".

- **Measurement** - The idea that amplitudes are negative as long as "no one is looking" and "collapse" (by squaring them) to positive probabilities when they are *measured* is deeply disturbing. Indeed, as shown by EPR and Bell, this leads to various strange outcomes such as "spooky actions at a distance", where we can create correlations between the results of measurements in places far removed. Unfortunately (or fortunately?) these strange outcomes have been confirmed experimentally.

- **Entanglement** - The notion that two parts of the system could be connected in this weird way where measuring one will affect the other is known as *quantum entanglement*.

As counter-intuitive as these concepts are, they have been experimentally confirmed, so we just have to live with them.

R

Remark 23.3 — More on quantum. The discussion in this lecture is quite brief and somewhat superficial. The chapter on quantum computation in my book with Arora (see draft here) is one relatively short resource that contains essentially everything we discuss here and more. See also this blog post of Aaronson for a high level explanation of Shor's algorithm which ends with links to several more detailed expositions. This lecture of Aaronson contains a great discussion of the feasibility of quantum computing (Aaronson's course lecture notes and the book that they spawned are fantastic reads as well). The videos of Umesh Vazirani'z EdX course are an accessible and recommended introduction to quantum computing. See the "bibliographical notes" section at the end of this chapter for more resources.

23.5 QUANTUM COMPUTING AND COMPUTATION - AN EXECUTIVE SUMMARY.

One of the strange aspects of the quantum-mechanical picture of the world is that unlike in the billiard ball example, there is no obvious algorithm to simulate the evolution of n particles over t time periods in $poly(n,t)$ steps. In fact, the natural way to simulate n quantum particles will require a number of steps that is *exponential* in n. This is a huge headache for scientists that actually need to do these calculations in practice.

In the 1981, physicist Richard Feynman proposed to "turn this lemon to lemonade" by making the following almost tautological observation:

If a physical system cannot be simulated by a computer in T steps, the system can be considered as performing a computation that would take more than T steps.

So, he asked whether one could design a quantum system such that its outcome y based on the initial condition x would be some function $y = f(x)$ such that **(a)** we don't know how to efficiently compute in any other way, and **(b)** is actually useful for something.[6] In 1985, David Deutsch formally suggested the notion of a quantum Turing machine, and the model has been since refined in works of Deutsch and Josza and Bernstein and Vazirani. Such a system is now known as a *quantum computer*.

For a while these hypothetical quantum computers seemed useful for one of two things. First, to provide a general-purpose mechanism to simulate a variety of the real quantum systems that people care about, such as various interactions inside molecules in quantum chemistry. Second, as a challenge to the *Extended Church Turing hypothesis* which says that every physically realizable computation device can be modeled (up to polynomial overhead) by Turing machines (or equivalently, NAND-TM / NAND-RAM programs).

Quantum chemistry is important (and in particular understanding it can be a bottleneck for designing new materials, drugs, and more), but it is still a rather niche area within the broader context of computing (and even scientific computing) applications. Hence for a while most researchers (to the extent they were aware of it), thought of quantum computers as a theoretical curiosity that has little bearing to practice, given that this theoretical "extra power" of quantum computer seemed to offer little advantage in the majority of the problems people want to solve in areas such as combinatorial optimization, machine learning, data structures, etc..

To some extent this is still true today. As far as we know, quantum computers, if built, will *not* provide exponential speed ups for 95% of the applications of computing.[7] In particular, as far as we know, quantum computers will *not* help us solve **NP** complete problems in polynomial or even sub-exponential time, though *Grover's algorithm* (Remark 23.4) does yield a quadratic advantage in many cases.

However, there is one cryptography-sized exception: In 1994 Peter Shor showed that quantum computers can solve the integer factoring and discrete logarithm problems in polynomial time. This result has captured the imagination of a great many people, and completely energized research into quantum computing. This is both because the hardness of these particular problems provides the foundations for securing such a huge part of our communications (and these days, our economy), and because it was a powerful demonstration that

[6] As its title suggests, Feynman's lecture was actually focused on the other side of simulating physics with a computer. However, he mentioned that as a "side remark" one could wonder if it's possible to simulate physics with a new kind of computer - a "quantum computer" which would "not [be] a Turing machine, but a machine of a different kind". As far as I know, Feynman did not suggest that such a computer could be useful for computations completely outside the domain of quantum simulation. Indeed, he was more interested in the question of whether quantum mechanics could be simulated by a classical computer.

[7] This "95 percent" is a figure of speech, but not completely so. At the time of this writing, cryptocurrency mining electricity consumption is estimated to use up at least 70Twh or 0.3 percent of the world's production, which is about 2 to 5 percent of the total energy usage for the computing industry. All the current cryptocurrencies will be broken by quantum computers. Also, for many web servers the TLS protocol (which is based on the current non-lattice based systems and would be completely broken by quantum computing) is responsible for about 1 percent of the CPU usage.

quantum computers could turn out to be useful for problems that a-priori seemd to have nothing to do with quantum physics.

As we'll discuss later, at the moment there are several intensive efforts to construct large scale quantum computers. It seems safe to say that, as far as we know, in the next five years or so there will not be a quantum computer large enough to factor, say, a 1024 bit number. On the other hand, it does seem quite likely that in the very near future there will be quantum computers which achieve *some* task exponentially faster than the best-known way to achieve the same task with a classical computer. When and if a quantum computer is built that is strong enough to break reasonable parameters of Diffie Hellman, RSA and elliptic curve cryptography is anybody's guess. It could also be a "self destroying prophecy" whereby the existence of a small-scale quantum computer would cause everyone to shift away to lattice-based crypto which in turn will diminish the motivation to invest the huge resources needed to build a large scale quantum computer.[8]

[8] Of course, given that we're still hearing of attacks exploiting "export grade" cryptography that was supposed to disappear in the 1990's, I imagine that we'll still have products running 1024 bit RSA when everyone has a quantum laptop.

R

Remark 23.4 — Quantum computing and NP. Despite popular accounts of quantum computers as having variables that can take "zero and one at the same time" and therefore can "explore an exponential number of possibilities simultaneously", their true power is much more subtle and nuanced. In particular, as far as we know, quantum computers do *not* enable us to solve **NP** complete problems such as 3SAT in polynomial or even sub-exponential time. However, Grover's search algorithm does give a more modest advantage (namely, quadratic) for quantum computers over classical ones for problems in **NP**. In particular, due to Grover's search algorithm, we know that the k-SAT problem for n variables can be solved in time $O(2^{n/2}poly(n))$ on a quantum computer for every k. In contrast, the best known algorithms for k-SAT on a classical computer take roughly $2^{(1-\frac{1}{k})n}$ steps.

23.6 QUANTUM SYSTEMS

Before we talk about *quantum* computing, let us recall how we physically realize "vanilla" or *classical* computing. We model a *logical bit* that can equal 0 or a 1 by some physical system that can be in one of two states. For example, it might be a wire with high or low voltage, charged or uncharged capacitor, or even (as we saw) a pipe with or without a flow of water, or the presence or absence of a soldier crab. A *classical* system of n bits is composed of n such "basic systems", each of which can be in either a "zero" or "one" state. We can model the

state of such a system by a string $s \in \{0,1\}^n$. If we perform an operation such as writing to the 17-th bit the NAND of the 3rd and 5th bits, this corresponds to applying a *local* function to s such as setting $s_{17} = 1 - s_3 \cdot s_5$.

In the *probabilistic* setting, we would model the state of the system by a *distribution*. For an individual bit, we could model it by a pair of non-negative numbers α, β such that $\alpha + \beta = 1$, where α is the probability that the bit is zero and β is the probability that the bit is one. For example, applying the *negation* (i.e., NOT) operation to this bit corresponds to mapping the pair (α, β) to (β, α) since the probability that $NOT(\sigma)$ is equal to 1 is the same as the probability that σ is equal to 0. This means that we can think of the NOT function as the linear map $N : \mathbb{R}^2 \to \mathbb{R}^2$ such that $N \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}$ or equivalently as the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

If we think of the n-bit system as a whole, then since the n bits can take one of 2^n possible values, we model the state of the system as a vector p of 2^n probabilities. For every $s \in \{0,1\}^n$, we denote by e_s the 2^n dimensional vector that has 1 in the coordinate corresponding to s (identifying it with a number in $[2^n]$), and so can write p as $\sum_{s \in \{0,1\}^n} p_s e_s$ where p_s is the probability that the system is in the state s.

Applying the operation above of setting the 17-th bit to the NAND of the 3rd and 5th bits, corresponds to transforming the vector p to the vector Fp where $F : \mathbb{R}^{2^n} \to \mathbb{R}^{2^n}$ is the linear map that maps e_s to $e_{s_0 \cdots s_{16}(1-s_3 \cdot s_5)s_{18} \cdots s_{n-1}}$.[9]

[9] Since $\{e_s\}_{s \in \{0,1\}^n}$ is a *basis* for R^{2^n}, it sufficies to define the map F on vectors of this form.

Please make sure you understand why performing the operation will take a system in state p to a system in the state Fp. Understanding the evolution of probabilistic systems is a prerequisite to understanding the evolution of quantum systems.

If your linear algebra is a bit rusty, now would be a good time to review it, and in particular make sure you are comfortable with the notions of *matrices*, *vectors*, (orthogonal and orthonormal) *bases*, and *norms*.

23.6.1 Quantum amplitudes

In the quantum setting, the state of an individual bit (or "qubit", to use quantum parlance) is modeled by a pair of numbers (α, β) such that $|\alpha|^2 + |\beta|^2 = 1$. While in general these numbers can be *complex*, for the rest of this chapter, we will often assume they are

real (though potentially negative), and hence often drop the absolute value operator. (This turns out not to make much of a difference in explanatory power.) As before, we think of α^2 as the probability that the bit equals 0 and β^2 as the probability that the bit equals 1. As we did before, we can model the NOT operation by the map $N : \mathbb{R}^2 \to \mathbb{R}^2$ where $N(\alpha, \beta) = (\beta, \alpha)$.

Following quantum tradition, instead of using e_0 and e_1 as we did above, from now on we will denote the vector $(1, 0)$ by $|0\rangle$ and the vector $(0, 1)$ by $|1\rangle$ (and moreover, think of these as column vectors). This is known as the Dirac "ket" notation. This means that NOT is the unique linear map $N : \mathbb{R}^2 \to \mathbb{R}^2$ that satisfies $N|0\rangle = |1\rangle$ and $N|1\rangle = |0\rangle$. In other words, in the quantum case, as in the probabilistic case, NOT corresponds to the matrix

$$N = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} .$$

In classical computation, we typically think that there are only two operations that we can do on a single bit: keep it the same or negate it. In the quantum setting, a single bit operation corresponds to any linear map $OP : \mathbb{R}^2 \to \mathbb{R}^2$ that is *norm preserving* in the sense that for every α, β, if we apply OP to the vector $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ then we obtain a vector $\begin{pmatrix} \alpha' \\ \beta' \end{pmatrix}$ such that $\alpha'^2 + \beta'^2 = \alpha^2 + \beta^2$. Such a linear map OP corresponds to a unitary two by two matrix.[10] Keeping the bit the same corresponds to the matrix $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and (as we've seen) the NOT operations corresponds to the matrix $N = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. But there are other operations we can use as well. One such useful operation is the *Hadamard* operation, which corresponds to the matrix

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} +1 & +1 \\ +1 & -1 \end{pmatrix} .$$

In fact it turns out that Hadamard is all that we need to add to a classical universal basis to achieve the full power of quantum computing.

[10] As we mentioned, quantum mechanics actually models states as vectors with *complex* coordinates. However, this does not make any qualitative difference to our discussion.

23.6.2 Recap

The *state* of a *quantum system* of n qubits is modeled by an 2^n dimensional vector ψ of unit norm (i.e., squares of all coordinates sums up to 1), which we write as $\psi = \sum_{x \in \{0,1\}^n} \psi_x |x\rangle$ where $|x\rangle$ is the column vector that has 0 in all coordinates except the one corresponding

to x (identifying $\{0,1\}^n$ with the numbers $\{0,\ldots,2^n-1\}$). We use the convention that if a,b are strings of lengths k and ℓ respectively then we can write the $2^{k+\ell}$ dimensional vector with 1 in the ab-th coordinate and zero elsewhere not just as $|ab\rangle$ but also as $|a\rangle|b\rangle$. In particular, for every $x \in \{0,1\}^n$, we can write the vector $|x\rangle$ also as $|x_0\rangle|x_1\rangle\cdots|x_{n-1}\rangle$. This notation satisfies certain nice distributive laws such as $|a\rangle(|b\rangle+|b'\rangle)|c\rangle = |abc\rangle + |ab'c\rangle$.

A *quantum operation* on such a system is modeled by a $2^n \times 2^n$ *unitary matrix* U (one that satisfies $UU^\top = I$ where $U^\top$ is the *transpose* operation, or conjugate transpose for complex matrices). If the system is in state ψ and we apply to it the operation U, then the new state of the system is $U\psi$.

When we *measure* an n-qubit system in a state $\psi = \sum_{x\in\{0,1\}^n} \psi_x|x\rangle$, then we observe the value $x \in \{0,1\}^n$ with probability $|\psi_x|^2$. In this case, the system *collapses* to the state $|x\rangle$.

23.7 ANALYSIS OF BELL'S INEQUALITY (OPTIONAL)

Now that we have the notation in place, we can show a strategy for Alice and Bob to display "quantum telepathy" in Bell's Game. Recall that in the classical case, Alice and Bob can succeed in the "Bell Game" with probability at most $3/4 = 0.75$. We now show that quantum mechanics allows them to succeed with probability at least 0.8.[11]

[11] The strategy we show is not the best one. Alice and Bob can in fact succeed with probability $\cos^2(\pi/8) \sim 0.854$.

Lemma 23.5 There is a 2-qubit quantum state $\psi \in \mathbb{C}^4$ so that if Alice has access to the first qubit of ψ, can manipulate and measure it and output $a \in \{0,1\}$ and Bob has access to the second qubit of ψ and can manipulate and measure it and output $b \in \{0,1\}$ then $\Pr[a \oplus b = x \wedge y] \geq 0.8$.

Proof. Alice and Bob will start by preparing a 2-qubit quantum system in the state

$$\psi = \tfrac{1}{\sqrt{2}}|00\rangle + \tfrac{1}{\sqrt{2}}|11\rangle$$

(this state is known as an EPR pair). Alice takes the first qubit of the system to her room, and Bob takes the qubit to his room. Now, when Alice receives x if $x=0$ she does nothing and if $x=1$ she applies the unitary map $R_{-\pi/8}$ to her qubit where $R_\theta = \begin{pmatrix} cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$ is the unitary operation corresponding to rotation in the plane with angle θ. When Bob receives y, if $y=0$ he does nothing and if $y=1$ he applies the unitary map $R_{\pi/8}$ to his qubit. Then each one of them measures their qubit and sends this as their response.

Recall that to win the game Bob and Alice want their outputs to be more likely to differ if $x=y=1$ and to be more likely to agree

otherwise. We will split the analysis in one case for each of the four possible values of x and y.

Case 1: $x = 0$ and $y = 0$. If $x = y = 0$ then the state does not change. Because the state ψ is proportional to $|00\rangle + |11\rangle$, the measurements of Bob and Alice will always agree (if Alice measures 0 then the state collapses to $|00\rangle$ and so Bob measures 0 as well, and similarly for 1). Hence in the case $x = y = 0$, Alice and Bob always win.

Case 2: $x = 0$ and $y = 1$. If $x = 0$ and $y = 1$ then after Alice measures her bit, if she gets 0 then the system collapses to the state $|00\rangle$, in which case after Bob performs his rotation, his qubit is in the state $\cos(\pi/8)|0\rangle + \sin(\pi/8)|1\rangle$. Thus, when Bob measures his qubit, he will get 0 (and hence agree with Alice) with probability $\cos^2(\pi/8) \geq 0.85$. Similarly, if Alice gets 1 then the system collapses to $|11\rangle$, in which case after rotation Bob's qubit will be in the state $-\sin(\pi/8)|0\rangle + \cos(\pi/8)|1\rangle$ and so once again he will agree with Alice with probability $\cos^2(\pi/8)$.

The analysis for **Case 3**, where $x = 1$ and $y = 0$, is completely analogous to Case 2. Hence Alice and Bob will agree with probability $\cos^2(\pi/8)$ in this case as well.[12]

[12] We are using the (not too hard) observation that the result of this experiment is the same regardless of the order in which Alice and Bob apply their rotations and measurements.

Case 4: $x = 1$ and $y = 1$. For the case that $x = 1$ and $y = 1$, after both Alice and Bob perform their rotations, the state will be proportional to

$$R_{-\pi/8}|0\rangle R_{\pi/8}|0\rangle + R_{-\pi/8}|1\rangle R_{\pi/8}|1\rangle \ . \tag{23.1}$$

Intuitively, since we rotate one state by 45 degrees and the other state by -45 degrees, they will become orthogonal to each other, and the measurements will behave like independent coin tosses that agree with probability 1/2. However, for the sake of completeness, we now show the full calculation.

Opening up the coefficients and using $\cos(-x) = \cos(x)$ and $\sin(-x) = -\sin(x)$, we can see that (23.1) is proportional to

$$\begin{aligned} &\cos^2(\pi/8)|00\rangle + \cos(\pi/8)\sin(\pi/8)|01\rangle \\ -&\sin(\pi/8)\cos(\pi/8)|10\rangle + \sin^2(\pi/8)|11\rangle \\ &-\sin^2(\pi/8)|00\rangle + \sin(\pi/8)\cos(\pi/8)|01\rangle \\ -&\cos(\pi/8)\sin(\pi/8)|10\rangle + \cos^2(\pi/8)|11\rangle \ . \end{aligned}$$

Using the trigonometric identities $2\sin(\alpha)\cos(\alpha) = \sin(2\alpha)$ and $\cos(\alpha) - \sin^2(\alpha) = \cos(2\alpha)$, we see that the probability of getting any one of $|00\rangle, |10\rangle, |01\rangle, |11\rangle$ is proportional to $\cos(\pi/4) = \sin(\pi/4) = \frac{1}{\sqrt{2}}$. Hence all four options for (a, b) are equally likely, which mean that in this case $a = b$ with probability 0.5.

Taking all the four cases together, the overall probability of winning the game is at least $\frac{1}{4} \cdot 1 + \frac{1}{2} \cdot 0.85 + \frac{1}{4} \cdot 0.5 = 0.8$.

■

R

Remark 23.6 — Quantum vs probabilistic strategies. It is instructive to understand what about quantum mechanics enabled this gain in Bell's Inequality. Consider the following analogous probabilistic strategy for Alice and Bob. They agree that each one of them will output 0 if they get 0 as input and output 1 with probability p if they get 1 as input. In this case one can see that their success probability would be $\frac{1}{4}\cdot 1 + \frac{1}{2}(1-p) + \frac{1}{4}[2p(1-p)] = 0.75 - 0.5p^2 \leq 0.75$. The quantum strategy we described above can be thought of as a variant of the probabilistic strategy for parameter p set to $\sin^2(\pi/8) = 0.15$. But in the case $x = y = 1$, instead of disagreeing only with probability $2p(1-p) = 1/4$, we can use the negative probabilities in the quantum world and rotate the state in opposite directions. Therefore, the probability of disagreement ends up being $\sin^2(\pi/4) = 0.5$.

23.8 QUANTUM COMPUTATION

Recall that in the classical setting, we modeled computation as obtained by a sequence of *basic operations*. We had two types of computational models:

- *Non uniform models of computation* such as Boolean circuits and NAND-CIRC programs, where a finite function $f : \{0,1\}^n \to \{0,1\}$ is computable in size T if it can be expressed as a combination of T basic operations (gates in a circuit or lines in a NAND-CIRC program)
- *Uniform models of computation* such as Turing machines and NAND-TM programs, where an infinite function $F : \{0,1\}^* \to \{0,1\}$ is computable in time $T(n)$ if there is a single algorithm that on input $x \in \{0,1\}^n$ evaluates $F(x)$ using at most $T(n)$ basic steps.

When considering *efficient computation*, we defined the class **P** to consist of all infinite functions $F : \{0,1\}^* \to \{0,1\}$ that can be computed by a Turing machine or NAND-TM program in time $p(n)$ for some polynomial $p(\cdot)$. We defined the class $\mathbf{P}_{/\mathbf{poly}}$ to consists of all infinite functions $F : \{0,1\}^* \to \{0,1\}$ such that for every n, the restriction $F_{\restriction n}$ of F to $\{0,1\}^n$ can be computed by a Boolean circuit or NAND-CIRC program of size at most $p(n)$ for some polynomial $p(\cdot)$.

We will do the same for *quantum computation*, focusing mostly on the *non uniform* setting of quantum circuits, since that is simpler, and already illustrates the important differences with classical computing.

23.8.1 Quantum circuits

A *quantum circuit* is analogous to a Boolean circuit, and can be described as a directed acyclic graph. One crucial difference that the *out degree* of every vertex in a quantum circuit is at most one. This is because we cannot "reuse" quantum states without *measuring* them (which collapses their "probabilities"). Therefore, we cannot use the same qubit as input for two different gates.[13] Another more technical difference is that to express our operations as unitary matrices, we will need to make sure all our gates are *reversible*. This is not hard to ensure. For example, in the quantum context, instead of thinking of *NAND* as a (non reversible) map from $\{0,1\}^2$ to $\{0,1\}$, we will think of it as the reversible map on *three* qubits that maps a, b, c to $a, b, c \oplus NAND(a,b)$ (i.e., flip the last bit if *NAND* of the first two bits is 1). Equivalently, the NAND operation corresponds to the 8×8 unitary matrix U_{NAND} such that (identifying $\{0,1\}^3$ with $[8]$) for every $a, b, c \in \{0,1\}$, if $|abc\rangle$ is the basis element with 1 in the abc-th coordinate and zero elsewhere, then $U_{NAND}|abc\rangle = |ab(c \oplus NAND(a,b))\rangle$.[14] If we order the rows and columns as $000, 001, 010, \ldots, 111$, then U_{NAND} can be written as the following matrix:

$$U_{NAND} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

If we have an n qubit system, then for $i, j, k \in [n]$, we will denote by $U^{i,j,k}_{NAND}$ as the $2^n \times 2^n$ unitary matrix that corresponds to applying U_{NAND} to the i-th, j-th, and k-th bits, leaving the others intact. That is, for every $v = \sum_{x \in \{0,1\}^n} v_x |x\rangle$, $U^{i,j,k}_{NAND} v = \sum_{x \in \{0,1\}^n} v_x |x_0 \cdots x_{k-1}(x_k \oplus NAND(x_i, x_j))x_{k+1} \cdots x_{n-1}\rangle$.

As mentioned above, we will also use the *Hadamard* or *HAD* operation. A *quantum circuit* is obtained by applying a sequence of U_{NAND} and *HAD* gates, which correspond to the matrix

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} +1 & +1 \\ +1 & -1 \end{pmatrix} .$$

Another way to define H is that for $b \in \{0,1\}$, $H|b\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}(-1)^b|1\rangle$. We define HAD^i to be the $2^n \times 2^n$ unitary matrix that applies *HAD* to the i-th qubit and leaves the others intact. Using the

[13] This is known as the No Cloning Theorem.

[14] Readers familiar with quantum computing should note that U_{NAND} is a close variant of the so called Toffoli gate and so QNAND-CIRC programs correspond to quantum circuits with the Hadamard and Toffoli gates.

ket notation, we can write this as

$$HAD^i \sum_{x\in\{0,1\}^n} v_x|x\rangle = \tfrac{1}{\sqrt{2}} \sum_{x\in\{0,1\}^n} |x_0\cdots x_{i-1}\rangle\left(|0\rangle + (-1)^{x_i}|1\rangle\right)|x_i\cdots x_{n-1}\rangle \ .$$

A *quantum circuit* is obtained by composing these basic operations on some m qubits. If $m \geq n$, we use a circuit to compute a function $f : \{0,1\}^n \to \{0,1\}$:

- On input x, we initialize the system to hold $x_0, \ldots, x_{n-1}$ in the first n qubits, and initialize all remaining $m-n$ qubits to zero.
- We execute each elementary operation one by one.
- At the end of the computation, we *measure* the system, and output the result of the last qubit (i.e. the qubit in location $m-1$).[15]
- We say that the circuit *computes* f, if the probability that this output equals $f(x)$ is at least $2/3$. Note that this probability is obtained by summing up the squares of the amplitudes of all coordinates in the final state of the system corresponding to vectors $|y\rangle$ where $y_{m-1} = f(x)$.

[15] For simplicity we restrict attention to functions with a single bit of output, though the definition of quantum circuits naturally extends to circuits with multiple outputs.

Formally this is defined as follows:

Definition 23.7 — Quantum circuit. A *quantum circuit of m inputs and s gates* over the $\{U_{NAND}, HAD\}$ basis is a sequence of s unitary $2^n \times 2^n$ matrices $U_0, \ldots, U_{s-1}$ such that each matrix U_ℓ is either of the form $NAND^{i,j,k}$ for $i,j,k \in [n]$ or HAD^i for $i \in [n]$.

A quantum circuit *computes* a function $f : \{0,1\}^n \to \{0,1\}$ if the following is true for every $x \in \{0,1\}^n$:

Let v be the vector

$$v = U_{s-1}U_{s-2}\cdots U_1U_0|x0^{m-n}\rangle$$

and write v as $\sum_{y\in\{0,1\}^m} v_y|y\rangle$. Then

$$\sum_{y\in\{0,1\}^m \text{ s.t. } y_{m-1}=f(x)} |v_y|^2 \geq \frac{2}{3} \ .$$

P Please stop here and see that this definition makes sense to you.

Once we have the notion of quantum circuits, we can define the quantum analog of $\mathbf{P}_{/\mathbf{poly}}$ (i.e., define the class of functions computable by *polynomial size quantum circuits*) as follows:

Definition 23.8 — $\mathbf{BQP}_{/\mathbf{poly}}$. Let $F : \{0,1\}^* \to \{0,1\}$. We say that $F \in \mathbf{BQP}_{/\mathbf{poly}}$ if there exists some polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $n \in \mathbb{N}$, if $F_{\restriction n}$ is the restriction of F to inputs of length n, then there is a quantum circuit of size at most $p(n)$ that computes $F_{\restriction n}$.

R

Remark 23.9 — The obviously exponential fallacy. A priori it might seem "obvious" that quantum computing is exponentially powerful, since to perform a quantum computation on n bits we need to maintain the 2^n dimensional state vector and apply $2^n \times 2^n$ matrices to it. Indeed popular descriptions of quantum computing (too) often say something along the lines that the difference between quantum and classical computer is that a classical bit can either be zero or one while a qubit can be in both states at once, and so in many qubits a quantum computer can perform exponentially many computations at once.

Depending on how you interpret it, this description is either false or would apply equally well to *probabilistic computation*, even though we've already seen that every randomized algorithm can be simulated by a similar-sized circuit, and in fact we conjecture that $\mathbf{BPP} = \mathbf{P}$.

Moreover, this "obvious" approach for simulating a quantum computation will take not just exponential time but *exponential space* as well, while it can be shown that using a simple recursive formula one can calculate the final quantum state using *polynomial space* (in physics this is known as "Feynman path integrals"). So, the exponentially long vector description by itself does not imply that quantum computers are exponentially powerful. Indeed, we cannot *prove* that they are (i.e., as far as we know, every QNAND-CIRC program could be simulated by a NAND-CIRC program with polynomial overhead), but we do have some problems (integer factoring most prominently) for which they do provide exponential speedup over the currently best *known* classical (deterministic or probabilistic) algorithms.

23.8.2 QNAND-CIRC programs (optional)

Just like in the classical case, there is an equivalence between circuits and straightline programs, and so we can define the programming language QNAND that is the quantum analog of our NAND-CIRC programming language. To do so, we only add a single operation: `HAD(foo)` which applies the single-bit operation H to the variable `foo`.

We also use the following interpretation to make NAND reversible: `foo = NAND(bar,blah)` means that we modify `foo` to be the XOR of its original value and the NAND of `bar` and `blah`. (In other words, apply the 8 by 8 unitary transformation U_{NAND} defined above to the three qubits corresponding to `foo`, `bar` and `blah`.) If `foo` is initialized to zero then this makes no difference.

If P is a QNAND-CIRC program with n input variables, ℓ workspace variables, and m output variables, then running it on the input $x \in \{0,1\}^n$ corresponds to setting up a system with $n+m+\ell$ qubits and performing the following process:

1. We initialize the input variables `X[0]` ... `X[`$n-1$`]` to $x_0, \ldots, x_{n-1}$ and all other variables to 0.

2. We execute the program line by line, applying the corresponding physical operation H or U_{NAND} to the qubits that are referred to by the line.

3. We *measure* the output variables `Y[0]`, ..., `Y[`$m-1$`]` and output the result (if there is more than one output then we measure more variables).

23.8.3 Uniform computation

Just as in the classical case, we can define *uniform* computational models. For example, we can define the *QNAND-TM programming language* to be QNAND augmented with loops and arrays just like NAND-TM is obtained from NAND. Using this we can define the class **BQP** which is the uniform analog of $\mathbf{BQP}_{/\mathbf{poly}}$. Just as in the classical setting it holds that $\mathbf{BPP} \subseteq \mathbf{P}_{/\mathbf{poly}}$, in the quantum setting it can be shown that $\mathbf{BQP} \subseteq \mathbf{BQP}_{/\mathbf{poly}}$. Just like the classical case, we can also use Quantum Turing Machines instead of QNAND-TM to define **BQP**.

Yet another way to define **BQP** is the following: a function $F : \{0,1\}^* \rightarrow \{0,1\}$ is in **BQP** if **(1)** $F \in \mathbf{BQP}_{/\mathbf{poly}}$ and **(2)** moreover for every n, the quantum circuit that verifies this can be generated by a *classical polynomial time NAND-TM program* (or, equivalently, a polynomial-time Turing machine).[16] We use this definition here, though an equivalent one can be made using QNAND-TM or quantum Turing machines:

Definition 23.10 — The class BQP. Let $F : \{0,1\}^* \rightarrow \{0,1\}$. We say that $F \in \mathbf{BQP}$ if there exists a polynomial time NAND-TM program P such that for every n, $P(1^n)$ is the description of a quantum circuit C_n that computes the restriction of F to $\{0,1\}^n$.

[16] This is analogous to the alternative characterization of **P** that appears in **??**.

P

One way to verify that you've understood these definitions it to see that you can prove (1) **P** $\subseteq$ **BQP** and in fact the stronger statement **BPP** $\subseteq$ **BQP**, (2) **BQP** $\subseteq$ **EXP**, and (3) For every **NP**-complete function F, if $F \in$ **BQP** then **NP** $\subseteq$ **BQP**. Exercise 23.1 asks you to work these out.

The relation between **NP** and **BQP** is not known (see also Remark 23.4). It is widely believed that **NP** $\not\subseteq$ **BQP**, but there is no consensus whether or not **BQP** $\subseteq$ **NP**. It is quite possible that these two classes are *incomparable*, in the sense that **NP** $\not\subseteq$ **BQP** (and in particular no **NP**-complete function belongs to **BQP**) but also **BQP** $\not\subseteq$ **NP** (and there are some interesting candidates for such problems).

It can be shown that *QNANDEVAL* (evaluating a quantum circuit on an input) is computable by a polynomial size QNAND-CIRC program, and moreover this program can even be generated *uniformly* and hence *QNANDEVAL* is in **BQP**. This allows us to "port" many of the results of classical computational complexity into the quantum realm as well.

R

Remark 23.11 — Restricting attention to circuits. Because the non uniform model is a little cleaner to work with, in the rest of this chapter we mostly restrict attention to this model, though all the algorithms we discuss can be implemented in uniform computation as well.

23.9 PHYSICALLY REALIZING QUANTUM COMPUTATION

To realize quantum computation one needs to create a system with n independent binary states (i.e., "qubits"), and be able to manipulate small subsets of two or three of these qubits to change their state. While by the way we defined operations above it might seem that one needs to be able to perform arbitrary unitary operations on these two or three qubits, it turns out that there several choices for *universal sets* - a small constant number of gates that generate all others. The biggest challenge is how to keep the system from being measured and *collapsing* to a single classical combination of states. This is sometimes known as the *coherence time* of the system. The threshold theorem says that there is some absolute constant level of errors τ so that if errors are created at every gate at rate smaller than τ then we can recover from those and perform arbitrary long computations. (Of course there are different ways to model the errors and so there are actually several threshold *theorems* corresponding to various noise models).

There have been several proposals to build quantum computers:

- Superconducting quantum computers use super-conducting electric circuits to do quantum computation. These are currently the devices with largest number of fully controllable qubits.
- At Harvard, Lukin's group is using cold atoms to implement quantum computers.
- Trapped ion quantum computers use the states of an ion to simulate a qubit. People have made some recent advances on these computers too. For example, an ion-trap computer was used to implement Shor's algorithm to factor 15. (It turns out that $15 = 3 \times 5$:))
- Topological quantum computers use a different technology, which is more stable by design but arguably harder to manipulate to create quantum computers.

These approaches are not mutually exclusive and it could be that ultimately quantum computers are built by combining all of them together. At the moment, we have devices with about 100 qubits, and about 1% error per gate. Such restricted machines are sometimes called "Noisy Intermediate-Scale Quantum Computers" or "NISQ". See this article by John Preskil for some of the progress and applications of such more restricted devices. If the number of qubits is increased and the error is decreased by one or two orders of magnitude, we could start seeing more applications.

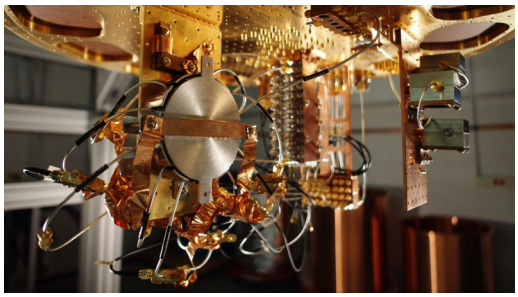


Figure 23.3: Superconducting quantum computer prototype at Google. Image credit: Google / MIT Technology Review.

23.10 SHOR'S ALGORITHM: HEARING THE SHAPE OF PRIME FACTORS

Bell's Inequality is a powerful demonstration that there is something very strange going on with quantum mechanics. But could this "strangeness" be of any use to solve computational problems not directly related to quantum systems? A priori, one could guess the answer is *no*. In 1994 Peter Shor showed that one would be wrong:

> **Theorem 23.12 — Shor's Algorithm.** There is a polynomial-time quantum algorithm that on input an integer M (represented in base two), outputs the prime factorization of M.

Another way to state Theorem 23.12 is that if we define $FACTORING : \{0,1\}^* \to \{0,1\}$ to be the function that on input a pair of numbers (M, X) outputs 1 if and only if M has a factor P such that $2 \leq P \leq X$, then *FACTORING* is in **BQP**. This is an exponential improvement over the best known classical algorithms, which take

roughly $2^{\tilde{O}(n^{1/3})}$ time, where the $\tilde{O}$ notation hides factors that are polylogarithmic in n. While we will not prove Theorem 23.12 in this chapter, we will sketch some of the ideas behind the proof.

23.10.1 Period finding

At the heart of Shor's Theorem is an efficient quantum algorithm for finding *periods* of a given function. For example, a function $f : \mathbb{R} \to \mathbb{R}$ is *periodic* if there is some $h > 0$ such that $f(x + h) = f(x)$ for every x (e.g., see Fig. 23.4).

Musical notes yield one type of periodic function. When you pull on a string on a musical instrument, it vibrates in a repeating pattern. Hence, if we plot the speed of the string (and so also the speed of the air around it) as a function of time, it will correspond to some *periodic* function. The length of the period is known as the *wave length* of the note. The *frequency* is the number of times the function repeats itself within a unit of time. For example, the "Middle C" note has a frequency of 261.63 Hertz, which means its period is $1/(261.63)$ seconds.

If we play a *chord* by playing several notes at once, we get a more complex periodic function obtained by combining the functions of the individual notes (see Fig. 23.5). The human ear contains many small hairs, each of which is sensitive to a narrow band of frequencies. Hence when we hear the sound corresponding to a chord, the hairs in our ears actually separate it out to the components corresponding to each frequency.

It turns out that (essentially) *every* periodic function $f : \mathbb{R} \to \mathbb{R}$ can be decomposed into a sum of simple *wave* functions (namely functions of the form $x \mapsto \sin(\theta x)$ or $x \mapsto \cos(\theta x)$). This is known as the Fourier Transform (see Fig. 23.6). The Fourier transform makes it easy to compute the period of a given function: it will simply be the least common multiple of the periods of the constituent waves.

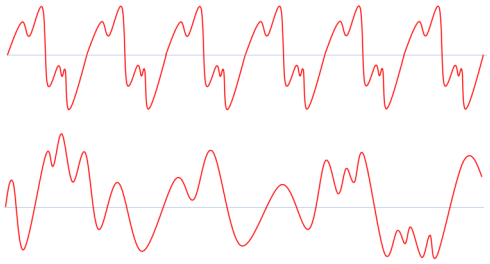


Figure 23.4: Top: A periodic function. Bottom: An a-periodic function.

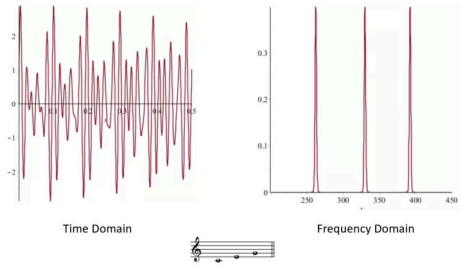


Figure 23.5: Left: The air-pressure when playing a "C Major" chord as a function of time. Right: The coefficients of the Fourier transform of the same function, we can see that it is the sum of three freuencies corresponding to the C, E and G notes (261.63, 329.63 and 392 Hertz respectively). Credit: Bjarke Mønsted's Quora answer.

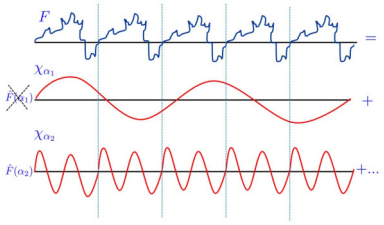


Figure 23.6: If f is a periodic function then when we represent it in the Fourier transform, we expect the coefficients corresponding to wavelengths that do not evenly divide the period to be very small, as they would tend to "cancel out".

23.10.2 Shor's Algorithm: A bird's eye view

On input a an integer M, Shor's algorithm outputs the prime factorization of M in time that is polynomial in $\log M$. The main steps in the algorithm are the following:

Step 1: Reduce to period finding. The first step in the algorithm is to pick a random $A \in \{0, 1 \ldots, M-1\}$ and define the function $F_A : \{0,1\}^m \to \{0,1\}^m$ as $F_A(x) = A^x \pmod M$ where we identify the string $x \in \{0,1\}^m$ with an integer using the binary representation, and similarly represent the integer $A^x \pmod M$ as a string. (We will choose m to be some polynomial in m and so in particular $\{0,1\}^m$ is a large enough set to represent all the numbers in $\{0, 1, \ldots, M-1\}$).

Some not-too-hard (though somewhat technical) calculations show that: **(1)** The function F_A is *periodic* (i.e., there is some integer p_A such that $F_A(x+p_A) = F_A(x)$ for almost[17] every x) and more importantly **(2)** If we can recover the period p_A of F_A for several randomly chosen A's, then we can recover the factorization of M. Hence, factoring M reduces to finding out the period of the function F_A. Exercise 23.2 asks you to work out this for the related task of computing the *discrete logarithm* (which underlies the security of the Diffie-Hellman key exchange and elliptic curve cryptography).

[17] We'll ignore this "almost" qualifier in the discussion below. It causes some annoying, yet ultimately manageable, technical issues in the full-fledged algorithm.

Step 2: Period finding via the Quantum Fourier Transform. Using a simple trick known as "repeated squaring", it is possible to compute the map $x \mapsto F_A(x)$ in time polynomial in m, which means we can also compute this map using a polynomial number of NAND gates,and so in particular we can generate in polynomial quantum time a quantum state ρ that is (up to normalization) equal to

$$\sum_{x \in \{0,1\}^m} |x\rangle|F_A(x)\rangle \quad .$$

In particular, if we were to *measure* the state ρ, we would get a random pair of the form (x,y) where $y = F_A(x)$. So far, this is not at all impressive. After all, we did not need the power of quantum computing to generate such pairs: we could simply generate a random x and then compute $F_A(x)$.

Another way to describe the state ρ is that the coefficient of $|x\rangle|y\rangle$ in ρ is proportional to $f_{A,y}(x)$ where $f_{A,y} : \{0,1\}^m \to \mathbb{R}$ is the function such that

$$f_{A,y}(x) = \begin{cases} 1 & y = A^x (\mod M) \\ 0 & \text{otherwise} \end{cases} .$$

The magic of Shor's algorithm comes from a procedure known as the *Quantum Fourier Transform*. It allows to change the state ρ into the state $\hat{\rho}$ where the coefficient of $|x\rangle|y\rangle$ is now proportional to the *x-th Fourier coefficient* of $f_{A,y}$. In other words, if we measure the state $\hat{\rho}$, we will obtain a pair (x,y) such that the probability of choosing x is proportional to the square of the weight of the *frequency* x in the representation of the function $f_{A,y}$. Since for every y, the function $f_{A,y}$ has the period p_A, it can be shown that the frequency x will be (almost[18]) a multiple of p_A. If we make several such samples $y_0, \ldots, y_k$ and obtain the frequencies $x_1, \ldots, x_k$, then the true period p_A divides all of them, and it can be shown that it is going to be in fact the *greatest common divisor* (g.c.d.) of all these frequencies: a value which can be computed in polynomial time.

[18] The "almost" qualifier again appears because the original function was only "almost" periodic, but it turns out this can be handled by using an "approximate greatest common divisor" algorithm instead of a standard g.c.d. below. The latter can be obtained using a tool known as the continued fraction representation of a number.

As mentioned above, we can recover the factorization of M from the periods of $F_{A_0}, \ldots, F_{A_t}$ for some randomly chosen $A_0, \ldots, A_t$ in $\{0, \ldots, M-1\}$ and t which is polynomial in $\log M$.

The resulting algorithm can be described in a high (and somewhat inaccurate) level as follows:

Shor's Algorithm: (*sketch*)

Input: Number $M \in \mathbb{N}$.

Output: Prime factorization of M.

Operations:

1. Repeat the following $k = poly(\log M)$ number of times:
   a. Choose $A \in \{0, \ldots, M-1\}$ at random, and let $f_A : \mathbb{Z}_M \to \mathbb{Z}_M$ be the map $x \mapsto A^x \mod M$.
   b. For $t = poly(\log M)$, repeat t times the following step: *Quantum Fourier Transform* to create a quantum state $|\psi\rangle$ over $poly(\log(m))$ qubits, such that if we measure $|\psi\rangle$ we obtain a pair of strings (j, y) with probability proportional to the square of the coefficient corresponding to the wave function $x \mapsto \cos(x\pi j/M)$ or $x \mapsto \sin(x\pi j/M)$ in the Fourier transform of the function $f_{A,y} : \mathbb{Z}_m \to \{0,1\}$ defined as $f_{A,y}(x) = 1$ iff $f_A(x) = y$.
   c. If $j_1, \ldots, j_t$ are the coefficients we obtained in the previous step, then the least common multiple of $M/j_1, \ldots, M/j_t$ is likely to be the *period* of the function f_A.
2. If we let $A_0, \ldots, A_{k-1}$ and $p_0, \ldots, p_{k-1}$ be the numbers we chose in the previous step and the corresponding periods of the functions $f_{A_0}, \ldots, f_{A_{k-1}}$ then we can use classical results in number theory to obtain from these a non-trivial prime factor Q of M (if such exists). We can now run the algorithm again with the (smaller) input M/Q to obtain all other factors.

Reducing factoring to order finding is cummbersome, but can be done in polynomial time using a classical computer. The key quantum ingredient in Shor's algorithm is the *quantum fourier transform*.

R

Remark 23.13 — Quantum Fourier Transform. Despite its name, the Quantum Fourier Transform does *not* actually give a way to compute the Fourier Transform of a function $f : \{0,1\}^m \to \mathbb{R}$. This would be impossible to do in time polynomial in m, as simply writing down the Fourier Transform would require 2^m coefficients. Rather the Quantum Fourier Transform gives a *quantum state* where the amplitude corresponding to an element (think: frequency) h is equal to the corresponding Fourier coefficient. This allows to sample from a distribution where h is drawn with probability proportional to the square of its Fourier coefficient. This is not the same as computing the

Fourier transform, but is good enough for recovering the period.

23.11 QUANTUM FOURIER TRANSFORM (ADVANCED, OPTIONAL)

The above description of Shor's algorithm skipped over the implementation of the main quantum ingredient: the *Quantum Fourier Transform* algorithm. In this section we discuss the ideas behind this algorithm. We will be rather brief and imprecise. Remark 23.3 and Section 23.13 contain references to sources of more information about this topic.

To understand the Quantum Fourier Transform, we need to better understand the Fourier Transform itself. In particular, we will need to understand how it applies not just to functions whose input is a real number but to functions whose domain can be any arbitrary commutative *group*. Therefore we now take a short detour to (very basic) *group theory*, and define the notion of periodic functions over groups.

R

Remark 23.14 — Group theory. While we define the concepts we use, some background in group or number theory might be quite helpful for fully understanding this section.

We will not use anything more than the basic properties of finite Abelian groups. Specifically we use the following notions:

- A finite *group* $\mathbb{G}$ can be thought of as simply a set of elements and some *binary operation* $\star$ on these elements (i.e., if $g, h \in \mathbb{G}$ then $g \star h$ is an element of $\mathbb{G}$ as well).
- The operation $\star$ satisfies the sort of properties that a product operation does, namely, it is *associative* (i.e., $(g \star h) \star f = g \star (h \star f)$) and there is some element 1 such that $g \star 1 = g$ for all g, and for every $g \in \mathbb{G}$ there exists an element g^{-1} such that $g \star g^{-1} = 1$.
- A group is called *commutative* (also known as *Abelian*) if $g \star h = h \star g$ for all $g, h \in \mathbb{G}$.

The Fourier transform is a deep and vast topic, on which we will barely touch upon here. Over the real numbers, the Fourier transform of a function f is obtained by expressing f in the form $\sum \hat{f}(\alpha)\chi_\alpha$ where the χ_α's are "wave functions" (e.g. sines and cosines). However, it turns out that the same notion exists for *every* Abelian group $\mathbb{G}$.

Specifically, for every such group $\mathbb{G}$, if f is a function mapping $\mathbb{G}$ to $\mathbb{C}$, then we can write f as

$$f = \sum_{g \in \mathbb{G}} \hat{f}(g)\chi_g \quad , \tag{23.2}$$

where the χ_g's are functions mapping $\mathbb{G}$ to $\mathbb{C}$ that are analogs of the "wave functions" for the group $\mathbb{G}$ and for every $g \in \mathbb{G}$, $\hat{f}(g)$ is a complex number known as the *Fourier coefficient of f corresponding to g*.[19] The representation (23.2) is known as the *Fourier expansion* or *Fourier transform* of f, the numbers $(\hat{f}(g))_{g\in\mathbb{G}}$ are known as the *Fourier coefficients* of f and the functions $(\chi_g)_{g\in\mathbb{G}}$ are known as the *Fourier characters*. The central property of the Fourier characters is that they are *homomorphisms* of the group into the complex numbers, in the sense that for every $x, x' \in \mathbb{G}$, $\chi_g(x \star x') = \chi_g(x)\chi_g(x')$, where $\star$ is the group operation. One corollary of this property is that if $\chi_g(h) = 1$ then χ_g is h *periodic* in the sense that $\chi_g(x \star h) = \chi_g(x)$ for every x. It turns out that if f is periodic with minimal period h, then the only Fourier characters that have non zero coefficients in the expression (23.2) are those that are h periodic as well. This can be used to recover the period of f from its Fourier expansion.

[19] The equation (23.2) means that if we think of f as a $|\mathbb{G}|$ dimensional vector over the complex numbers, then we can write this vector as a sum (with certain coefficients) of the vectors $\{\chi_g\}_{g\in\mathbb{G}}$.

23.11.1 Quantum Fourier Transform over the Boolean Cube: Simon's Algorithm

We now describe the simplest setting of the Quantum Fourier Transform: the group $\{0,1\}^n$ with the XOR operation, which we'll denote by $(\{0,1\}^n, \oplus)$. It can be shown that the Fourier transform over $(\{0,1\}^n, \oplus)$ corresponds to expressing $f : \{0,1\}^n \to \mathbb{C}$ as

$$f = \sum_{y \in \{0,1\}} \hat{f}(y)\chi_y$$

where $\chi_y : \{0,1\}^n \to \mathbb{C}$ is defined as $\chi_y(x) = (-1)^{\sum_i y_i x_i}$ and $\hat{f}(y) = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} f(x)(-1)^{\sum_i y_i x_i}$.

The Quantum Fourier Transform over $(\{0,1\}^n, \oplus)$ is actually qutie simple:

Theorem 23.15 — QFT Over the Boolean Cube. Let $\rho = \sum_{x\in\{0,1\}^n} f(x)|x\rangle$ be a quantum state where $f : \{0,1\}^n \to \mathbb{C}$ is some function satisfying $\sum_{x\in\{0,1\}^n} |f(x)|^2 = 1$. Then we can use n gates to transform ρ to the state

$$\sum_{y\in\{0,1\}^n} \hat{f}(y)|y\rangle$$

where $f = \sum_y \hat{f}(y)\chi_y$ and $\chi_y : \{0,1\}^n \to \mathbb{C}$ is the function $\chi_y(x) = -1^{\sum x_i y_i}$.

Proof Idea:

The idea behind the proof is that the *Hadamard* operation corresponds to the *Fourier transform* over the group $\{0,1\}^n$ (with the XOR operations). To show this, we just need to do the calculations.

⋆

Proof of Theorem 23.15. We can express the Hadamard operation *HAD* as follows:

$$HAD|a\rangle = \tfrac{1}{\sqrt{2}}(|0\rangle + (-1)^a|1\rangle) \ .$$

We are given the state

$$\rho = \sum_{x\in\{0,1\}^n} f(x)|x\rangle \ .$$

Now suppose that we apply the *HAD* operation to each of the n qubits. We can see that we get the state

$$2^{-n/2} \sum_{x\in\{0,1\}^n} f(x) \prod_{i=0}^{n-1} (|0\rangle + (-1)^{x_i}|1\rangle) \ .$$

We can now use the distributive law and open up a term of the form

$$f(x)(|0\rangle + (-1)^{x_0}|1\rangle) \cdots (|0\rangle + (-1)^{x_{n-1}}|1\rangle)$$

to the following sum over 2^n terms:[20]

$$f(x) \sum_{y\in\{0,1\}^n} (-1)^{\sum y_i x_i}|y\rangle \ .$$

But by changing the order of summations, we see that the final state is

$$\sum_{y\in\{0,1\}^n} 2^{-n/2} \Big(\sum_{x\in\{0,1\}^n} f(x)(-1)^{\sum x_i y_i} \Big) |y\rangle$$

which exactly corresponds to $\hat{\rho}$.

■

[20] If you find this confusing, try to work out why $(|0\rangle + (-1)^{x_0}|1\rangle)(|0\rangle + (-1)^{x_1}|1\rangle)(|0\rangle + (-1)^{x_2}|1\rangle)$ is the same as the sum over 2^3 terms $|000\rangle + (-1)^{x_2}|001\rangle + \cdots + (-1)^{x_0+x_1+x_2}|111\rangle$.

23.11.2 From Fourier to Period finding: Simon's Algorithm (advanced, optional)

Using Theorem 23.15 it is not hard to get an algorithm that can recover a string $h^* \in \{0,1\}^n$ given a circuit that computes a function $F : \{0,1\}^n \to \{0,1\}^*$ that is h^* *periodic* in the sense that $F(x) = F(x')$ for

distinct x, x' if and only if $x' = x \oplus h^*$. The key observation is that if we compute the state $\sum_{x\in\{0,1\}^n} |x\rangle|F(x)\rangle$, and perform the Quantum Fourier transform on the first n qubits, then we would get a state such that the only basis elements with nonzero coefficients would be of the form $|y\rangle$ where

$$\sum y_i h_i^* = 0(\mod 2) \tag{23.3}$$

So, by measuring the state, we can obtain a sample of a random y satisfying (23.3). But since (23.3) is a *linear* equation modulo 2 about the unknown n variables $h_0^*, \ldots, h_{n-1}^*$, if we repeat this procedure to get n such equations, we will have at least as many equations as variables and (it can be shown that) this will suffice to recover h^*.

This result is known as Simon's Algorithm, and it preceded and inspired Shor's algorithm.

23.11.3 From Simon to Shor (advanced, optional)

Theorem 23.15 seemed to really use the special bit-wise structure of the group $\{0,1\}^n$, and so one could wonder if it can be extended to other groups. However, it turns out that we can in fact achieve such a generalization.

The key step in Shor's algorithm is to implement the Fourier transform for the group $\mathbb{Z}_L$ which is the set of numbers $\{0, \ldots, L-1\}$ with the operation being addition modulo L. In this case it turns out that the Fourier characters are the functions $\chi_y(x) = \omega^{yx}$ where $\omega = e^{2\pi i/L}$ (i here denotes the complex number $\sqrt{-1}$). The y-th Fourier coefficient of a function $f : \mathbb{Z}_L \to \mathbb{C}$ is

$$\hat{f}(y) = \tfrac{1}{\sqrt{L}} \sum_{x\in\mathbb{Z}_L} f(x)\omega^{xy} \,. \tag{23.4}$$

The key to implementing the Quantum Fourier Transform for such groups is to use the same recursive equations that enable the classical Fast Fourier Transform (FFT) algorithm. Specifically, consider the case that $L = 2^\ell$. We can separate the sum over x in (23.4) to the terms corresponding to even x's (of the form $x = 2z$) and odd x's (of the form $x = 2z+1$) to obtain

$$\hat{f}(y) = \tfrac{1}{\sqrt{L}} \sum_{z\in Z_{L/2}} f(2z)(\omega^2)^{yz} + \tfrac{\omega^y}{\sqrt{L}} \sum_{z\in\mathbb{Z}_{L/2}} f(2z+1)(\omega^2)^{yz} \tag{23.5}$$

which reduces computing the Fourier transform of f over the group $\mathbb{Z}_{2^\ell}$ to computing the Fourier transform of the functions f_{even} and f_{odd} (corresponding to the applying f to only the even and odd x's respectively) which have $2^{\ell-1}$ inputs that we can identify with the group $\mathbb{Z}_{2^{\ell-1}} = \mathbb{Z}_{L/2}$.

Specifically, the Fourier characters of the group $\mathbb{Z}_{L/2}$ are the functions $\chi_y(x) = e^{2\pi i/(L/2)yx} = (\omega^2)^{yx}$ for every $x, y \in \mathbb{Z}_{L/2}$. Moreover, since $\omega^L = 1$, $(\omega^2)^y = (\omega^2)^{y \mod L/2}$ for every $y \in \mathbb{N}$. Thus (23.5) translates into

$$\hat{f}(y) = \hat{f}_{even}(y \mod L/2) + \omega^y \hat{f}_{odd}(y \mod L/2) \ .$$

This observation is usually used to obtain a fast (e.g. $O(L \log L)$) time to compute the Fourier transform in a classical setting, but it can be used to obtain a quantum circuit of $poly(\log L)$ gates to transform a state of the form $\sum_{x \in \mathbb{Z}_L} f(x)|x\rangle$ to a state of the form $\sum_{y \in \mathbb{Z}_L} \hat{f}(y)|y\rangle$.

The case that L is not an exact power of two causes some complications in both the classical case of the Fast Fourier Transform and the quantum setting of Shor's algorithm. However, it is possible to handle these. The idea is that we can embed Z_L in the group $\mathbb{Z}_{A \cdot L}$ for any integer A, and we can find an integer A such that $A \cdot L$ will be close enough to a power of 2 (i.e., a number of the form 2^m for some m), so that if we do the Fourier transform over the group $\mathbb{Z}_{2^m}$ then we will not introduce too many errors.

Chapter Recap

- The state of an n-qubit quantum system can be modeled as a 2^n dimensional vector.
- An operation on the state corresponds to applying a unitary matrix to this vector.
- Quantum circuits are obtained by composing basic operations such as *HAD* and U_{NAND}.
- We can use quantum circuits to define the classes $\mathbf{BQP}_{/\mathbf{poly}}$ and **BQP**, which are the quantum analogs of $\mathbf{P}_{/\mathbf{poly}}$ and **BPP** respectively.
- There are some problems for which the best known quantum algorithm is *exponentially faster* than the best known, but quantum computing is not a panacea. In particular, as far as we know, quantum computers could still require exponential time to solve **NP**-complete problems such as *SAT*.

23.12 EXERCISES

Remark 23.16 — Disclaimer. Most of the exercises have been written in the summer of 2018 and haven't yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exer-

cise, and optionally complement this with an email to me with more details about the attempted solution.

Exercise 23.1 — Quantum and classical complexity class relations. Prove the following relations between quantum complexity classes and classical ones:

1. $\mathbf{P}_{/\mathbf{poly}} \subseteq \mathbf{BQP}_{/\mathbf{poly}}$.[21]
2. $\mathbf{P} \subseteq \mathbf{BQP}$.[22]
3. $\mathbf{BPP} \subseteq \mathbf{BQP}$.[23]
4. $\mathbf{BQP} \subseteq \mathbf{EXP}$.[24]
5. If $SAT \in \mathbf{BQP}$ then $\mathbf{NP} \subseteq \mathbf{BQP}$.[25]

[21] *Hint:* You can use U_{NAND} to simulate NAND gates.

[22] *Hint:* Use the alternative characterization of **P** as in **??**.

[23] *Hint:* You can use the *HAD* gate to simulate a coin toss.

[24] *Hint:* In exponential time simulating quantum computation boils down to matrix multiplication.

[25] *Hint:* If a reduction can be implemented in **P** it can be implemented in **BQP** as well.

■

Exercise 23.2 — Discrete logarithm from order finding. Show a probabilistic polynomial time classical algorithm that given an Abelian finite group $\mathbb{G}$ (in the form of an algorithm that computes the group operation), a *generator* g for the group, and an element $h \in \mathbb{G}$, as well access to a black box that on input $f \in \mathbb{G}$ outputs the *order* of f (the smallest a such that $f^a = 1$), computes the *discrete logarithm* of h with respect to g. That is the algorithm should output a number x such that $g^x = h$. See footnote for hint.[26]

[26] We are given $h = g^x$ and need to recover x. To do so we can compute the order of various elements of the form $h^a g^b$. The order of such an element is a number c satisfying $c(xa + b) = 0 \pmod{|\mathbb{G}|}$. With a few random examples we will get a non trivial equation on x (where c is not zero modulo $|\mathbb{G}|$) and then we can use our knowledge of a, b, c to recover x.

■

23.13 BIBLIOGRAPHICAL NOTES

Chapters 9 and 10 in the book *Quantum Computing Since Democritus* give an informal but highly informative introduction to the topics of this lecture and much more. Shor's and Simon's algorithms are also covered in Chapter 10 of my book with Arora on computational complexity.

There are many excellent videos available online covering some of these materials. The Fourier transform is covered in this videos of Dr. Chris Geoscience, Clare Zhang and Vi Hart. More specifically to quantum computing, the videos of Umesh Vazirani on the Quantum Fourier Transform and Kelsey Houston-Edwards on Shor's Algorithm are very recommended.

Chapter 10 in Avi Wigderson's book gives a high level overview of quantum computing. Andrew Childs' lecture notes on quantum algorithms, as well as the lecture notes of Umesh Vazirani, John Preskill, and John Watrous

Regarding quantum mechanics in general, this video illustrates the double slit experiment, this Scientific American video is a nice exposition of Bell's Theorem. This talk and panel moderated by Brian Greene discusses some of the philosophical and technical issues around quantum mechanics and its so called "measurement problem". The Feynmann lecture on the Fourier Transform and quantum mechanics in general are very much worth reading.

The Fast Fourier Transform, used as a component in Shor's algorithm, is one of the most useful algorithms across many applications areas. The stories of its discovery by Gauss in trying to calculate asteroid orbits and rediscovery by Tukey during the cold war are fascinating as well.

23.14 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

23.15 ACKNOWLEDGEMENTS

Thanks to Scott Aaronson for many helpful comments about this chapter.

VI

APPENDICES

Bibliography

[Aar05] Scott Aaronson. "NP-complete problems and physical reality". In: *ACM Sigact News* 36.1 (2005). Available on https://arxiv.org/abs/quant-ph/0502072, pp. 30–52.

[Aar13] Scott Aaronson. *Quantum computing since Democritus*. Cambridge University Press, 2013.

[Aar16] Scott Aaronson. "P =? NP". In: *Open problems in mathematics*. Available on https://www.scottaaronson.com/papers/pnp.pdf. Springer, 2016, pp. 1–122.

[Aar20] Scott Aaronson. "The Busy Beaver Frontier". In: *SIGACT News* (2020). Available on https://www.scottaaronson.com/papers/bb.pdf.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. "PRIMES is in P". In: *Annals of mathematics* (2004), pp. 781–793.

[Asp18] James Aspens. *Notes on Discrete Mathematics*. Online textbook for CS 202. Available on http://www.cs.yale.edu/homes/aspnes/classes/202/notes.pdf. 2018.

[Bar84] H. P. Barendregt. *The lambda calculus : its syntax and semantics*. Amsterdam New York New York, N.Y: North-Holland Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, 1984. ISBN: 0444875085.

[Bjo14] Andreas Bjorklund. "Determinant sums for undirected hamiltonicity". In: *SIAM Journal on Computing* 43.1 (2014), pp. 280–299.

[Blä13] Markus Bläser. *Fast Matrix Multiplication*. Graduate Surveys 5. Theory of Computing Library, 2013, pp. 1–60. DOI: 10.4086/toc.gs.2013.005. URL: http://www.theoryofcomputing.org/library.html.

[Boo47] George Boole. *The mathematical analysis of logic*. Philosophical Library, 1847.

[BU11] David Buchfuhrer and Christopher Umans. "The complexity of Boolean formula minimization". In: *Journal of Computer and System Sciences* 77.1 (2011), pp. 142–153.

[Bur78] Arthur W Burks. "Booke review: Charles S. Peirce, The new elements of mathematics". In: *Bulletin of the American Mathematical Society* 84.5 (1978), pp. 913–918.

[Cho56] Noam Chomsky. "Three models for the description of language". In: *IRE Transactions on information theory* 2.3 (1956), pp. 113–124.

[Chu41] Alonzo Church. *The calculi of lambda-conversion*. Princeton London: Princeton University Press H. Milford, Oxford University Press, 1941. ISBN: 978-0-691-08394-0.

[CM00] Bruce Collier and James MacLachlan. *Charles Babbage: And the engines of perfection*. Oxford University Press, 2000.

[Coh08] Paul Cohen. *Set theory and the continuum hypothesis*. Mineola, N.Y: Dover Publications, 2008. ISBN: 0486469212.

[Coh81] Danny Cohen. "On holy wars and a plea for peace". In: *Computer* 14.10 (1981), pp. 48–54.

[Coo66] SA Cook. "On the minimum computation time for multiplication". In: *Doctoral dissertation, Harvard University Department of Mathematics, Cambridge, Mass.* 1 (1966).

[Coo87] James W Cooley. "The re-discovery of the fast Fourier transform algorithm". In: *Microchimica Acta* 93.1-6 (1987), pp. 33–45.

[Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[CR73] Stephen A Cook and Robert A Reckhow. "Time bounded random access machines". In: *Journal of Computer and System Sciences* 7.4 (1973), pp. 354–375.

[CRT06] Emmanuel J Candes, Justin K Romberg, and Terence Tao. "Stable signal recovery from incomplete and inaccurate measurements". In: *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences* 59.8 (2006), pp. 1207–1223.

[CT06] Thomas M. Cover and Joy A. Thomas. "Elements of information theory 2nd edition". In: *Willey-Interscience: NJ* (2006).

[Dau90] Joseph Warren Dauben. *Georg Cantor: His mathematics and philosophy of the infinite*. Princeton University Press, 1990.

[De 47] Augustus De Morgan. *Formal logic: or, the calculus of inference, necessary and probable*. Taylor and Walton, 1847.

[Don06] David L Donoho. "Compressed sensing". In: *IEEE Transactions on information theory* 52.4 (2006), pp. 1289–1306.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008. ISBN: 978-0-07-352340-8.

[Ell10] Jordan Ellenberg. "Fill in the blanks: Using math to turn lo-res datasets into hi-res samples". In: *Wired Magazine* 18.3 (2010), pp. 501–509.

[Ern09] Elizabeth Ann Ernst. "Optimal combinational multi-level logic synthesis". Available on https://deepblue.lib.umich.edu/handle/2027.42/62373. PhD thesis. University of Michigan, 2009.

[Fle18] Margaret M. Fleck. *Building Blocks for Theoretical Computer Science*. Online book, available at http://mfleck.cs.illinois.edu/building-blocks/. 2018.

[Für07] Martin Fürer. "Faster integer multiplication". In: *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*. 2007, pp. 57–66.

[FW93] Michael L Fredman and Dan E Willard. "Surpassing the information theoretic bound with fusion trees". In: *Journal of computer and system sciences* 47.3 (1993), pp. 424–436.

[GKS17] Daniel Günther, Ágnes Kiss, and Thomas Schneider. "More efficient universal circuit constructions". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 443–470.

[Gom+08] Carla P Gomes et al. "Satisfiability solvers". In: *Foundations of Artificial Intelligence* 3 (2008), pp. 89–134.

[Gra05] Judith Grabiner. *The origins of Cauchy's rigorous calculus*. Mineola, N.Y: Dover Publications, 2005. ISBN: 9780486438153.

[Gra83] Judith V Grabiner. "Who gave you the epsilon? Cauchy and the origins of rigorous calculus". In: *The American Mathematical Monthly* 90.3 (1983), pp. 185–194.

[Hag98] Torben Hagerup. "Sorting and searching on the word RAM". In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 1998, pp. 366–398.

[Hal60] Paul R Halmos. *Naive set theory*. Republished in 2017 by Courier Dover Publications. 1960.

[Har41] GH Hardy. "A Mathematician's Apology". In: (1941).

[HJB85] Michael T Heideman, Don H Johnson, and C Sidney Burrus. "Gauss and the history of the fast Fourier transform". In: *Archive for history of exact sciences* 34.3 (1985), pp. 265–277.

[HMU14] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to automata theory, languages, and computation*. Harlow, Essex: Pearson Education, 2014. ISBN: 1292039051.

[Hof99] Douglas Hofstadter. *Gödel, Escher, Bach : an eternal golden braid*. New York: Basic Books, 1999. ISBN: 0465026567.

[Hol01] Jim Holt. "The Ada perplex: how Byron's daughter came to be celebrated as a cybervisionary". In: *The New Yorker* 5 (2001), pp. 88–93.

[Hol18] Jim Holt. *When Einstein walked with Gödel : excursions to the edge of thought*. New York: Farrar, Straus and Giroux, 2018. ISBN: 0374146705.

[HU69] John E Hopcroft and Jeffrey D Ullman. "Formal languages and their relation to automata". In: (1969).

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. ISBN: 0-201-02988-X.

[HV19] David Harvey and Joris Van Der Hoeven. "Integer multiplication in time O(n log n)". working paper or preprint. Mar. 2019. URL: https://hal.archives-ouvertes.fr/hal-02070778.

[Joh12] David S Johnson. "A brief history of NP-completeness, 1954–2012". In: *Documenta Mathematica* (2012), pp. 359–376.

[Juk12] Stasys Jukna. *Boolean function complexity: advances and frontiers*. Vol. 27. Springer Science & Business Media, 2012.

[Kar+97] David R. Karger et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*. 1997, pp. 654–663. DOI: 10.1145/258533.258660. URL: https://doi.org/10.1145/258533.258660.

[Kar95] ANATOLII ALEXEEVICH Karatsuba. "The complexity of computations". In: *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation* 211 (1995), pp. 169–183.

[Kle91] Israel Kleiner. "Rigor and Proof in Mathematics: A Historical Perspective". In: *Mathematics Magazine* 64.5 (1991), pp. 291–314. ISSN: 0025570X, 19300980. URL: http://www.jstor.org/stable/2690647.

[Kle99] Jon M Kleinberg. "Authoritative sources in a hyperlinked environment". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 604–632.

[Koz97] Dexter Kozen. *Automata and computability*. New York: Springer, 1997. ISBN: 978-3-642-85706-5.

[KT06] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006. ISBN: 978-0-321-37291-8.

[Kun18] Jeremy Kun. *A programmer's introduction to mathematics*. Middletown, DE: CreateSpace Independent Publishing Platform, 2018. ISBN: 1727125452.

[Liv05] Mario Livio. *The equation that couldn't be solved : how mathematical genius discovered the language of symmetry*. New York: Simon & Schuster, 2005. ISBN: 0743258207.

[LLM18] Eric Lehman, Thomson F. Leighton, and Albert R. Meyer. *Mathematics for Computer Science*. 2018.

[LMS16] Helger Lipmaa, Payman Mohassel, and Seyed Saeed Sadeghian. "Valiant's Universal Circuit: Improvements, Implementation, and Applications." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 17.

[Lup58] O Lupanov. "A circuit synthesis method". In: *Izv. Vuzov, Radiofizika* 1.1 (1958), pp. 120–130.

[Lup84] Oleg B. Lupanov. *Asymptotic complexity bounds for control circuits*. In Russian. MSU, 1984.

[Lus+08] Michael Lustig et al. "Compressed sensing MRI". In: *IEEE signal processing magazine* 25.2 (2008), pp. 72–82.

[Lüt02] Jesper Lützen. "Between Rigor and Applications: Developments in the Concept of Function in Mathematical Analysis". In: *The Cambridge History of Science*. Ed. by Mary JoEditor Nye. Vol. 5. The Cambridge History of Science. Cambridge University Press, 2002, pp. 468–487. DOI: 10.1017/CHOL9780521571999.026.

[LZ19] Harry Lewis and Rachel Zax. *Essential Discrete Mathematics for Computer Science*. Princeton University Press, 2019. ISBN: 9780691190617.

[Maa85] Wolfgang Maass. "Combinatorial lower bound arguments for deterministic and nondeterministic Turing machines". In: *Transactions of the American Mathematical Society* 292.2 (1985), pp. 675–693.

[MM11] Cristopher Moore and Stephan Mertens. *The nature of computation*. Oxford University Press, 2011.

[MP43] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[MU17] Michael Mitzenmacher and Eli Upfal. *Probability and computing: randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[Neu45] John von Neumann. "First Draft of a Report on the EDVAC". In: (1945). Reprinted in the IEEE Annals of the History of Computing journal, 1993.

[NS05] Noam Nisan and Shimon Schocken. *The elements of computing systems: building a modern computer from first principles*. MIT press, 2005.

[Pag+99] Lawrence Page et al. *The PageRank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[Pie02] Benjamin Pierce. *Types and programming languages*. Cambridge, Mass: MIT Press, 2002. ISBN: 0262162091.

[Ric53] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[Rog96] Yurii Rogozhin. "Small universal Turing machines". In: *Theoretical Computer Science* 168.2 (1996), pp. 215–240.

[Ros19] Kenneth Rosen. *Discrete mathematics and its applications*. New York, NY: McGraw-Hill, 2019. ISBN: 125967651x.

[RS59] Michael O Rabin and Dana Scott. "Finite automata and their decision problems". In: *IBM journal of research and development* 3.2 (1959), pp. 114–125.

[Sav98] John E Savage. *Models of computation*. Vol. 136. Available electronically at http://cs.brown.edu/people/jsavage/book/. Addison-Wesley Reading, MA, 1998.

[Sch05] Alexander Schrijver. "On the history of combinatorial optimization (till 1960)". In: *Handbooks in operations research and management science* 12 (2005), pp. 1–68.

[Sha38] Claude E Shannon. "A symbolic analysis of relay and switching circuits". In: *Electrical Engineering* 57.12 (1938), pp. 713–723.

[Sha79] Adi Shamir. "Factoring numbers in O (logn) arithmetic steps". In: *Information Processing Letters* 8.1 (1979), pp. 28–31.

[She03] Saharon Shelah. "Logical dreams". In: *Bulletin of the American Mathematical Society* 40.2 (2003), pp. 203–228.

[She13] Henry Maurice Sheffer. "A set of five independent postulates for Boolean algebras, with application to logical constants". In: *Transactions of the American mathematical society* 14.4 (1913), pp. 481–488.

[She16] Margot Shetterly. *Hidden figures : the American dream and the untold story of the Black women mathematicians who helped win the space race*. New York, NY: William Morrow, 2016. ISBN: 9780062363602.

[Sin97] Simon Singh. *Fermat's enigma : the quest to solve the world's greatest mathematical problem*. New York: Walker, 1997. ISBN: 0385493622.

[Sip97] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997. ISBN: 978-0-534-94728-6.

[Sob17] Dava Sobel. *The Glass Universe : How the Ladies of the Harvard Observatory Took the Measure of the Stars*. New York: Penguin Books, 2017. ISBN: 9780143111344.

[Sol14] Daniel Solow. *How to read and do proofs : an introduction to mathematical thought processes*. Hoboken, New Jersey: John Wiley & Sons, Inc, 2014. ISBN: 9781118164020.

[SS71] Arnold Schönhage and Volker Strassen. "Schnelle multiplikation grosser zahlen". In: *Computing* 7.3-4 (1971), pp. 281–292.

[Ste87] Dorothy Stein. *Ada : a life and a legacy*. Cambridge, Mass: MIT Press, 1987. ISBN: 0262691167.

[Str69] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische mathematik* 13.4 (1969), pp. 354–356.

[Swa02] Doron Swade. *The difference engine : Charles Babbage and the quest to build the first computer*. New York: Penguin Books, 2002. ISBN: 0142001449.

[Tho84] Ken Thompson. "Reflections on trusting trust". In: *Communications of the ACM* 27.8 (1984), pp. 761–763.

[Too63] Andrei L Toom. "The complexity of a scheme of functional elements realizing the multiplication of integers". In: *Soviet Mathematics Doklady*. Vol. 3. 4. 1963, pp. 714–716.

[Tur37] Alan M Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[Vad+12] Salil P Vadhan et al. "Pseudorandomness". In: *Foundations and Trends® in Theoretical Computer Science* 7.1–3 (2012), pp. 1–336.

[Val76] Leslie G Valiant. "Universal circuits (preliminary report)". In: *Proceedings of the eighth annual ACM symposium on Theory of computing*. ACM. 1976, pp. 196–203.

[Wan57] Hao Wang. "A variant to Turing's theory of computing machines". In: *Journal of the ACM (JACM)* 4.1 (1957), pp. 63–92.

[Weg87] Ingo Wegener. *The complexity of Boolean functions*. Vol. 1. BG Teubner Stuttgart, 1987.

[Wer74] PJ Werbos. "Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph. D. thesis, Harvard University, Cambridge, MA, 1974." In: (1974).

[Wig19] Avi Wigderson. *Mathematics and Computation*. Draft available on https://www.math.ias.edu/avi/book. Princeton University Press, 2019.

[Wil09] Ryan Williams. "Finding paths of length k in $O^*(2^k)$ time". In: *Information Processing Letters* 109.6 (2009), pp. 315–318.

[WN09] Damien Woods and Turlough Neary. "The complexity of small universal Turing machines: A survey". In: *Theoretical Computer Science* 410.4-5 (2009), pp. 443–450.

[WR12] Alfred North Whitehead and Bertrand Russell. *Principia mathematica*. Vol. 2. University Press, 1912.